# SANDIA REPORT

SAND2015-6778
Unlimited Release
Printed September 2015

# High Performance Computing - Power Application Programming Interface Specification Version 1.1a

James H. Laros III, David DeBonis, Ryan Grant, Suzanne M. Kelly,
Michael Levenhagen, Stephen Olivier, Kevin Pedretti

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

SAND2015-6778
Unlimited Release
Printed September 2015

# High Performance Computing - Power Application Programming Interface Specification Version 1.1a

James H. Laros III, David DeBonis, Ryan E. Grant,
Suzanne M. Kelly, Michael Levenhagen, Stephen Olivier, Kevin Pedretti
Center for Computing Research
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
jhlaros,ddeboni,regrant,smkelly,mjleven,slolivi,ktpedre @*sandia.gov*

**Abstract**

Measuring and controlling the power and energy consumption of high performance computing systems by various components in the software stack is an active research area [13, 3, 5, 10, 4, 21, 19, 16, 7, 17, 20, 18, 11, 1, 6, 14, 12]. Implementations in lower level software layers are beginning to emerge in some production systems, which is very welcome. To be most effective, a portable interface to measurement and control features would significantly facilitate participation by all levels of the software stack. We present a proposal for a standard power Application Programming Interface (API) that endeavors to cover the entire software space, from generic hardware interfaces to the input from the computer facility manager.

# Acknowledgment

Support for this work was provided through the Advanced Simulation and Computing (ASC) program funded by U.S. Department of Energy's National Nuclear Security Agency. We wish to thank our colleagues, Steve Hammond, Ryan Elmore, and Kris Munch at the National Renewable Energy Laboratory (NREL) for their contributions to the use case model which was the progenitor of this work. This effort was greatly enhanced by interactions with staff throughout Sandia as well as many external organizations.

Prior to the first open release of this specification a select group of individuals agreed to review an early draft of the specification and provide feedback. We would like to recognize the very significant contributions these individuals made and thank them for their time and efforts. The following individuals participated in an all day face-to-face review of the specification and provided written feedback (listed in alphabetical order): David Jackson (Adaptive Computing), Steve Martin (Cray), Indrani Paul (AMD), Phil Pokorny (Penguin Computing), Avi Purkayastha (National Renewable Energy Laboratory), Muralidhar Rajappa (Intel), and Jeff Stuecheli (IBM). The following individuals provided written feedback of the specification (listed in alphabetical order): Dorian Arnold (University of New Mexico), Natalie Bates (EEHPC), and Chung-Hsing Hsu (Oak Ridge National Laboratory). We hope to continue these important collaborations and develop new ones in an effort to represent and serve the HPC community as best we can.

# Contents

## Appendix

# Chapter 1

# Introduction

Achieving practical exascale supercomputing will require massive increases in energy efficiency. The bulk of this improvement will likely be derived from hardware advances such as improved semiconductor device technologies and tighter integration, hopefully resulting in more energy efficient computer architectures. Still, software will have an important role to play. With every generation of new hardware, more power measurement and control capabilities are exposed. Many of these features require software involvement to maximize feature benefits. This trend will allow algorithm designers to add power and energy efficiency to their optimization criteria. Similarly, at the system level, opportunities now exist for energy-aware scheduling to meet external utility constraints such as time of day cost charging and power ramp rate limitations. Finally, future architectures might not be able to operate all components at full capability for a range of reasons including temperature considerations or power delivery limitations. Software will need to make appropriate choices about how to allocate the available power budget given many, sometimes conflicting considerations.

For these reasons, we have developed a portable API for power measurement and control. This Power API provides multiple levels of abstractions to satisfy the requirements of multiple types of users [9]. The remainder of this document describes the details of this Power API specification.

## 1.1 Background

We draw our inspiration from efforts such as the MPI forum's[1] process. We seek to develop a de facto standard, led by a neutral national laboratory, which is funded by a neutral federal agency. Community involvement is critical to the effort. The laboratory team has been garnering participation by making presentations at workshops and operational group meetings. We desire community participation from university and other researchers, as well as HPC practitioners. Concurrent with the specification development, the authors are creating a reference implementation comprising a subset of the overall API functionality. This task is important to ensure that the specification is usable. The ultimate goal, however, is that vendors of the hardware and software components provide their own implementations. It is likely that some portion of these functions have already been written by vendors, but with slightly different calling arguments. For portability sake, we are

[1] http://www.mpi-forum.org

hopeful that the specific implementations can be melded to this proposed community API.

## 1.2 Motivation

The introductory paragraph above, offers a few examples where a Power API would be useful. This document's abstract provides references to a small subset of the current research activities that would benefit from a community-adopted power API. Additional, more fleshed out examples are described in the appendices of the *Power/Energy Use Cases for High Performance Computing* document [9]. To provide the proper mindset for reading this document, we offer the following list as well.

- A job is entering a checkpoint phase. The application requests a reduced processor frequency during the long I/O period.
- A developer is trying to understand frequency sensitivity of an algorithm and starts a tool that analyzes performance and power consumption while the job is running.
- Once an application's power signature is analyzed, future job submissions give power hints to the resource manager.
- A data center has a maximum of capacity of nn MW. One HPC system is down for extended maintenance. Other systems can have a higher maximum power cap.
- For electric bills based on peak usage periods, determine a maximum HPC load that minimizes loss of HPC use. Then direct the scheduler to enforce that peak usage.

## 1.3 Use Case Development

The *Power/Energy Use Cases for High Performance Computing* document [9] identifies the requirements for the Power API. Rather than a list, the requirements are specified as formal use cases employing the ISO/IEC 19501:2005 Unified Modeling Language (UML) standard, which is described in the reference manual by Booch, et al. [2]. While the term use case has come to be almost synonymous with scenario, the standard defines a use case *model*. The use case model does include scenario-like requirement specifications, but it also clearly identifies the roles and scope of the requirements. For this document, the key concepts from the use case model are *actor* and *system*. Each identified actor plays a distinct role in using the power API. Actors can be persons, other systems, or something else (e.g. cron, asynchronous event, etc.). For the Power API use case model, an HPC computer is broken down into logical systems. By breaking down the requirements into this use case model, we can clearly see the demarcation points requiring an API between external actors and each system. And by subsequently viewing systems as actors to the other systems, we obtain the complete set of necessary interfaces.

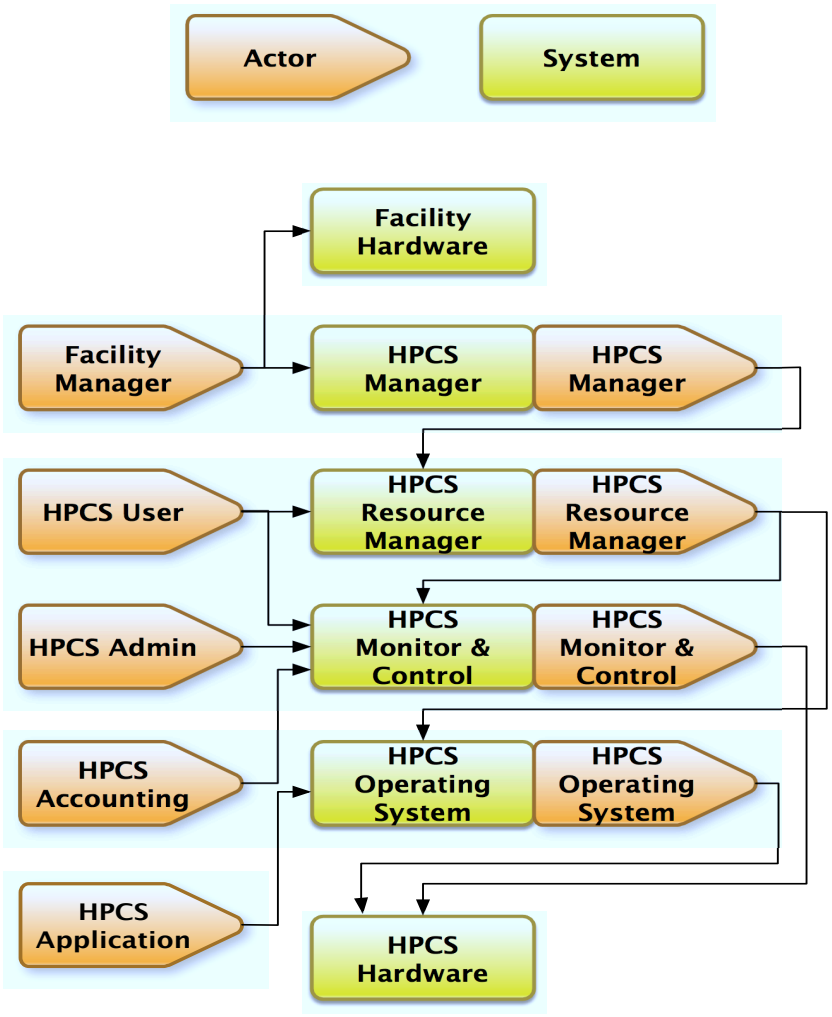


Figure 1.1: Top Level Conceptual Diagram representing the culmination of all Use Case Diagrams covered.

The specific actor/system pairs used for the power API are shown in Figure 1.1. The external actors are shown on the left portion of the diagram. Systems are shown as rectangles. The four systems conjoined with the actor symbol also serve as actors for some use cases. The ten sections within Chapter 6 provide function specifications for the ten actor/system pairs (Role/System pairs in the specification). The two missing interfaces are Facility Manager to Facility Hardware and Facility Manager to HPCS manager. These were included in the use case model to identify the boundaries of the specification and recognize important points of information input.

## 1.4 Security Model

The specification assumes traditional hardware (e.g. protection rings) and operating system support for access control. Implementations should only need traditional restrictions based on authenticated individual identity and/or the groups to which the individual belongs. A super user is likely needed as well. Depending on the implementation, the context structure (Section 3.3) may be sufficiently protected to allow for secure storage of access information. Future releases of the specification will address security and policy considerations in more detail.

# Chapter 2

# Theory of Operation

## 2.1 Overview

This section discusses many of the foundational concepts leveraged throughout the Power API specification. It should be noted that many terms commonly used when discussing object oriented languages are used in this section and the document as a whole. The use of these terms in no way implies that the Power API specification must be implemented using an object oriented language. We have attempted to achieve two goals, listed in order of priority: 1) programmer portability, where the programmer is the user of the API, and 2) the latitude of the implementor who will often become the user of the API benefitting from our first priority.

## 2.2 Power API Initialization

Using any of the Power API interfaces requires initialization. Initializaton returns a context. In the specification, the context is defined as an opaque pointer. This approach was taken to allow the maximum amount of flexibility to the implementor. The context returned will contain (act as the entry point to) the system description that is exposed to the user, all policy and privilege information, basically everything the user of the API requires to perform the functionality specified by the API. The system description is not required to be changed or updated during the life of a specific context. Initialization is accomplished by calling `PWR_CntxtInit()` (section 4.1).

## 2.3 Roles

The Power API specification leverages the concept of Roles. Roles represent the different types of users that exist which include:

- **Application** The application or application library executing on the compute resource. May also include run-time components running in user space.
- **Monitor and Control** Cluster management or Reliability Availability and Serviceability (RAS) systems, for example.

- **Operating System** Linux or specialized Light Weight Kernels which are found on HPC platforms and potentially portions of run-time systems.
- **User** The user of the HPC platform.
- **Resource Manager** This can include work load managers, schedulers, allocators and even portions of run-time systems.
- **Administrator** The system administrator or HPC platform manager.
- **HPCS Manager** The individual or individuals responsible for managing policy for the HPC platform, for example.
- **Accounting** Individual or software that produces reports of metrics for the HPC platform.

These brief definitions are not meant to be exhaustive. Roles are analogous with the *Actors* discussed in section 1.3. In some cases roles become the system that other roles interact with. For example, we specify an interface between the Application role (HPCS Application in figure 1.1) and the Operating System (HPCS Operating System in figure 1.1). The Operating System is the system (in UML terminology) that the Application role is interacting with. Notice in figure 1.1 that the specification also includes an interface between the Operating System role and the Hardware (HPCS Hardware in figure 1.1). These and other interfaces are described in chapter 6. The user of the API is required to specify what role they will assume when interacting with the system upon initialization of the API.

Roles are also provided as a mechanism for the implementation to express priority or precedence in circumstances where, for example, conflicting operations are requested.

## 2.4 System Description

The system description is the *view* of the system exposed to the user upon initialization via the context that is returned. Figure 2.1 depicts an example of a system description showing a hierarchical arrangement of objects. All object types listed in the specification must be defined by any implementation, but do not have to be used in the system description. The implementation chooses which objects will be employed in the system description and how they will be arranged. An object can only have a single parent but may have multiple children. Currently, a system description may only describe a single platform and have a single object of type `Platform` which represents the top of the hierarchy. Later revisions of the specification may include the ability to combine multiple platforms in the system description. This might be useful, for example, in representing an entire datacenter. While figure 2.1 depicts a homogeneous system description, homogeneity is *not* a requirement. In practice a system description can be heterogeneous and unbalanced.

To summarize the requirements:

- The `Platform` object type must be defined by the implementation and must appear at the top of the system description.
- All object types in this specification must be defined in any implementation. The use of the object types, with the exception of the `Platform` object type, is optional.

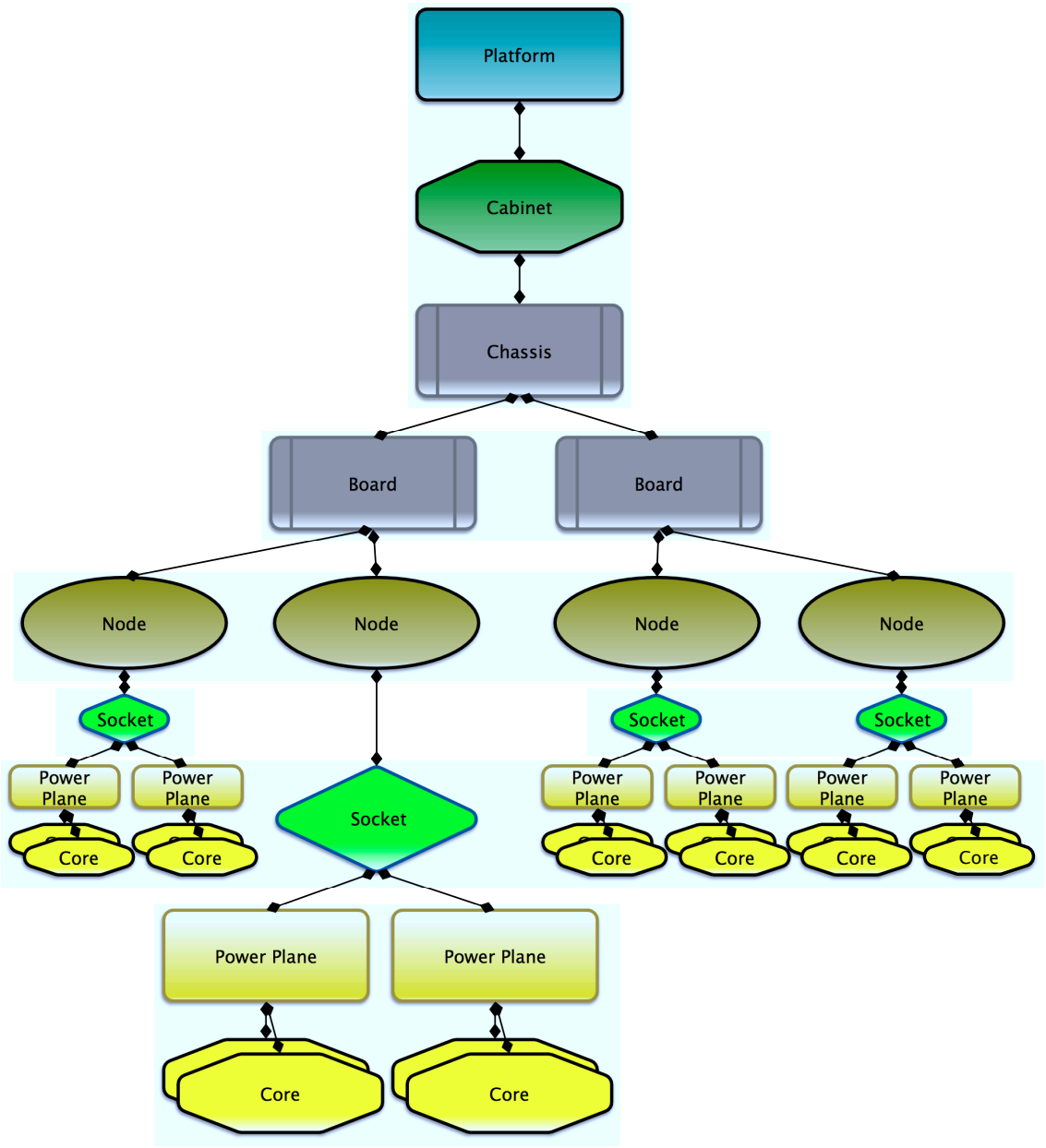


Figure 2.1: Hierarchical Depiction of System Objects

- Objects can only have one parent but may have many children. Currently the `Platform` object has no parent since it appears at the top of the system description. This will likely change in future versions of the specification.

The following is a list of the object types currently included in the specification along with a short description of each.

- Platform - Currently, the one and only Platform object is the top level object of the system description exposed to the user of the API. The Platform object is intended to conceptually represent the entire Platform. For example, if the Platform object has a power or energy measurement or control capability exposed through the Platform objects attributes the scope of these attributes should be platform wide.
- Cabinet - Objects of type Cabinet are intended to represent the cabinets or racks that act as enclosures (or logical groupings) for the platform equipment. Beyond the utility of convenient groups of lower level objects (equipment) cabinets may have power or energy relevant capabilities which can be exposed through attributes associated with each Cabinet object.
- Chassis - Objects of type Chassis are intended to be used for finer grained organization of objects within the higher level Cabinet object. Chassis, like cabinets may have power or energy relevant capabilities that can be exposed to the user.
- Board - Board objects offer another method of organization for underlying objects (equipment). Boards may also have power and or energy relevant capabilities which can be exposed through associated attributes. For example, a board could contain the power supply and the point of instrumentation for collecting power or energy samples for a node or multiple nodes.
- Node - The Node type is probably one of the most universally important object types. Measuring and controlling the power and or energy characteristics of a node or multiple nodes (grouped into multiple Boards, Chassis or Cabinets) is important for a many reasons and provides a wide range of flexibility of configuration to the implementor. For example, on HPC platforms a single application typically executes on many nodes. Understanding the energy use of an application run can be obtained by collecting the energy use (via the appropriate Node attribute) for each node participating in that application execution. Node objects will likely have many attributes exposing many power and energy relevant capabilities.
- Socket - The Socket object is intended to represent the one or more processor sockets, or other component types that can be thought of as sockets, that make up a Node. For example, a single Node object may be a dual socket (dual CPU) node. The implementor may choose to enclose other component types (a NIC for example) within a Socket object, or add other object types as they see fit to represent the architecture they are describing. They can also decide to omit the use of this, or any other object type (currently other than Platform) in the system description.
- Power Plane - The Power Plane object is used to organize lower level objects (any types of objects) within a power domain or single point of measurement and or control. For example, a pair of cores may share a power plane within a socket. This configuration is depicted in figure 2.1. This organization allows a pair of cores to be controlled from a single power control point in the hierarchy for convenience. This object type allows these power and energy relevant relationships to be expressed anywhere in the system description.
- Core - Core objects are intended to represent the individual processor cores within multi-

core CPUs (or possibly GPUs). Modern architectures have an increasing number of cores per CPU (or GPU). In the near future it is likely that an abstraction between Socket and core would become useful as the number of cores increase. Physical and logical groupings of cores already exist in current architectures.
- Memory - The Memory object type is included to represent the growing range of memory types that exist on HPC platforms. Individual cores, for example, have Memory in the form of cache which the implementor may choose to organize differently from the main memory of the Node or a tertiary level of memory such as NVRAM.
- NIC - The NIC object is intended to represent the Network Interface Controller. As with many other object types, the organization of a NIC in relation to Boards, Nodes or even Cores is architecture dependent. The NIC object type is included in hopes that there are power and energy relevant capabilities included in future NICs.

Additional object types may be defined by the implementor and placed anywhere in the hierarchy as long as the previously stated rules are not violated. Ultimately, the object types defined in this specification, and those added by the implementor, will be used to produce a system description describing the system presented to the user via the context returned upon initialization. Objects are used as interfaces to underlying functionality. The specification does not assume state is retained for objects. Additionally, the specification makes no guarantees with regards to race conditions between processes or threads.

## 2.5 Attributes

Attributes are an important part of the Power API. A large amount of basic functionality is exposed through the use of attributes. The term attribute is used somewhat conceptually since some attributes are implicit while others are explicitly defined as part of a required specification data structure (page 26). Attributes are used for a number of reasons such as to navigate through the system description, to access information or a measurement (sensor information for example) and for control (setting a P-state for example). Global attributes are attributes that are present for every object defined; whether required by the specification or added by the implementor.

The following is the list of global attributes:

- name - Unique identifying name of the object (see `PWR_ObjGetName` on page 38).
- entry point - The position in the hierarchy after initialization (see `PWR_CntxtGetEntryPoint` on page 37).
- type - The type of the object (see `PWR_ObjGetType` on page 38).
- parent - The parent of an object is the object that is above it in the hierarchy (see `PWR_ObjGetParent` on page 39). The only exception is the currently single platform object whose parent is a pointer to NULL.
- children - Object or objects directly below an object in the hierarchy (see `PWR_ObjGetChildren` on page 39).

Note, in the list above all the attributes are implicit. Explicit attributes are defined in `PWR_AttrName` (page 26). The majority of the attributes defined in the specification, and likely those added by an implementator, are, and will be, explicit. The implicit attributes defined above are primarily used for navigation and are accessed through attribute specific functions which are described in Section 4.2.

Explicit attributes are either accessed through the generic attribute interface (Section 4.4) or attribute specific functions found in either the section describing the specific interface in which they are used or in Chapter 4, *Core (Common) Interface Functions*.

The attribute interface is intended to keep the specification from growing every time additional functionality is either specified or added by an implementor. As long as the new functionality fits within the defined attribute interfaces no additional API functions are required to be specified.

## 2.6 Metadata

Each object and object attribute pair can have additional descriptive metadata associated with it. This information is often useful for getting a better understanding of the meaning of objects and attributes and how to interpret the values read from attributes. Examples include a human readable name and description strings, the list of values supported by an attribute, and measurement accuracy and precision. The metadata interface (see section 4.5) returns information relevant to either a specific object or a specific attribute of a specific object. A given attribute name may have different metadata for different objects, even if the objects are of the same type (e.g., the voltage attribute of two node objects may have different metadata accuracy values).

## 2.7 Thread Safety

Implementations of the Power API are not required to provide thread safety to multiple threads of the same process. If necessary, users of the Power API must use locking or some other mechanism to ensure that only one thread per process calls into the Power API at a time. This requirement only applies to threads of the same process that may issue conflicting operations. Different processes may make simultaneous Power API calls without any coordination. If thread concurrency within a process is required, the `PWR_CntxtInit` function can be called multiple times to initialize multiple Power API contexts. Multiple threads of the same process may then simultaneously call into the Power API, so long as each thread operates on a different Power API context. For example, a process with four threads may create four Power API contexts and associate one context with each thread. The threads may then make Power API calls without any additional coordination, so long as each thread operates only on its assigned context and the objects exposed by its assigned context. Threads should not operate on objects exposed by another thread's context without employing locking or some other coordination mechanism.

# Chapter 3

# Type Definitions

## 3.1 Opaque Types

The following type definitions are specified to be opaque pointers from the point of view of Power API users. Power API implementations will typically map these pointers to internal implementation-specific state. The reason for using opaque pointers is to hide non-portable implementation details from users and give implementors of the API maximum flexibility.

```
typedef void* PWR_Cntxt;
typedef void* PWR_Grp;
typedef void* PWR_Obj;
typedef void* PWR_Status;
typedef void* PWR_Stat;
```

## 3.2 Globally Relevant Definitions

The following definitions are specified on a global basis. The `PWR_MAJOR_VERSION` and `PWR_MINOR_VERSION` definitions are compile time constants that indicate the Power API version supported by the implementation. The `PWR_MAX_STRING_LEN` definition is a compile time constant that defines the maximum length of strings that can be returned from Power API calls, with the actual value being a vendor specific length.

```
#define PWR_MAJOR_VERSION 1
#define PWR_MINOR_VERSION 1
#define PWR_MAX_STRING_LEN VENDOR_MAX_STRING_LEN
```

## 3.3 Context Relevant Type Definitions

The `PWR_CntxtType` and `PWR_Role` types are required to be defined by all implementations of the Power API. When a new Power API context is created, one value from each of these types is used

to determine the kind of context created (see section 4.1). For PWR_CntxtType, the only required value that an implementation must define is PWR_CNTXT_DEFAULT. This indicates that the new context will only contain Power API functionality that is explicitly defined in the specification, with no implementation-specific extentions present. Implementors may extend PWR_CntxtType with additional values, such as PWR_CNTXT_VENDOR, to provide contexts with additional functionality.

We anticipate that most implementations of the Power API will define additional PWR_CntxtType values that provide additional functionality, such as vendor, platform, or model specific extentions. If an implementation extends the specification, the extensions should only be visible to the user when they use a context that was created with an implementation-specific PWR_CntxtType value. If the implementation-specific extensions are not available to the user, initialization using an implementation-specific PWR_CntxtType value should result in failure. The user must always be able to initialize a context using PWR_CNTXT_DEFAULT to to get a context containing only the standard specification features.

Differentiation between context types is the mechanism used by the Power API to enable extended vendor, platform or model specific capabilities while, at the same time, allowing portability for applications or tools that only leverage standard specification features. For example, a tool that leverages only the object and attribute types defined in the standard specification can initialize a Power API context using PWR_CNTXT_DEFAULT and not have to worry about dealing with any implementation-specific functionality. The context it receives will only provide functionality that is explicitly defined by the Power API specification.

PWR_Role is used to specify the role that the user is acting in when they initialize a new context. Additional roles may not be added by the implementor. Notice that there is a role defined for every actor in Chapter 6 - Role/Systems Interfaces. We intend that the user's role will serve many purposes, such as determining the view of the system that is provided within the context when combined with the system the user is acting on. Roles can also be used to help determine the privilege of the user's context for purposes such as resolving the precedence of conflicting operations.

**PWR_CntxtType**

```
typedef int PWR_CntxtType;
#define PWR_CNTXT_DEFAULT 0
#define PWR_CNTXT_VENDOR 1
```

### PWR_Role

```
typedef enum {
        PWR_ROLE_APP = 0, /* Application */
        PWR_ROLE_MC, /* Monitor and Control */
        PWR_ROLE_OS, /* Operating System */
        PWR_ROLE_USER, /* User */
        PWR_ROLE_RM, /* Resource Manager */
        PWR_ROLE_ADMIN, /* Administrator */
        PWR_ROLE_MGR, /* HPCS Manager */
        PWR_ROLE_ACC, /* Accounting */
        PWR_NUM_ROLES,
        /* */
        PWR_ROLE_INVALID = -1,
        PWR_ROLE_NOT_SPECIFIED = -2
} PWR_Role;
```

## 3.4 Object Relevant Type Definitions

The PWR_ObjType type is required to be defined by all implementations of the Power API specification. Objects with types defined by PWR_ObjType are used by the implementor to create the system description (see section 2.4) that is exposed to the user upon initialization. An implementation may extend this type by adding new object enumeration type, which must be added prior to PWR_NUM_OBJ_TYPES. The added implementation-specific object types will only be used by implementation-specific contexts (see section 3.3). Contexts that were initialized using the default context, PWR_CNTXT_DEFAULT, will only expose objects types defined in the list below.

### PWR_ObjType

```
typedef enum {
        PWR_OBJ_PLATFORM = 0,
        PWR_OBJ_CABINET,
        PWR_OBJ_CHASSIS,
        PWR_OBJ_BOARD,
        PWR_OBJ_NODE,
        PWR_OBJ_SOCKET,
        PWR_OBJ_CORE,
        PWR_OBJ_POWER_PLANE,
        PWR_OBJ_MEM,
        PWR_OBJ_NIC,
        PWR_NUM_OBJ_TYPES,
        /* */
        PWR_OBJ_INVALID = -1,
        PWR_OBJ_NOT_SPECIFIED = -2
} PWR_ObjType;
```

## 3.5 Attribute Relevant Type Definitions

The PWR_AttrName and PWR_AttrDataType types are required to be implemented. Both may be extended by the implementor and exposed using an implementation specified context type (see section 3.3). If new PWR_AttrName entries are added it is required that the attribute name is specified and commented as shown in the PWR_AttrName structure. Likewise, new types must be added to the PWR_AttrDataType structure. It's important to note that the attribute interface currently supports only numeric types. Attributes should only be added to this definition if they can be meaningfully supported by the attribute interface (section 4.4). Additional attributes must be added prior to PWR_NUM_ATTR_NAMES. The Attributes in PWR_AttrName expose what we consider foundational measurement and control interfaces. Additional capabilities are and can be added using additional operations and often interface specific functions.

The PWR_AttrAccessError type is used to hold the error returns that are popped from the PWR_Status handle (see section 3.1) using the PWR_StatusPopError function (see page 49).

### PWR_AttrName

```
typedef enum {
        PWR_ATTR_PSTATE = 0, /* uint64_t */
        PWR_ATTR_CSTATE, /* uint64_t */
        PWR_ATTR_CSTATE_LIMIT, /* uint64_t */
        PWR_ATTR_SSTATE, /* uint64_t */
        PWR_ATTR_CURRENT, /* double, amps */
        PWR_ATTR_VOLTAGE, /* double, volts */
        PWR_ATTR_POWER, /* double, watts */
        PWR_ATTR_POWER_LIMIT_MIN, /* double, watts */
        PWR_ATTR_POWER_LIMIT_MAX, /* double, watts */
        PWR_ATTR_FREQ, /* double, Hz */
        PWR_ATTR_FREQ_LIMIT_MIN, /* double, Hz */
        PWR_ATTR_FREQ_LIMIT_MAX, /* double, Hz */
        PWR_ATTR_ENERGY, /* double, joules */
        PWR_ATTR_TEMP, /* double, degrees Celsius */
        PWR_ATTR_OS_ID, /* uint64_t */
        PWR_ATTR_THROTTLED_TIME, /* uint64_t */
        PWR_ATTR_THROTTLED_COUNT, /* uint64_t */
        PWR_NUM_ATTR_NAMES,
        /* */
        PWR_ATTR_INVALID = -1,
        PWR_ATTR_NOT_SPECIFIED = -2
} PWR_AttrName;
```

**PWR_AttrDataType**

```
typedef enum {
        PWR_ATTR_DATA_DOUBLE = 0,
        PWR_ATTR_DATA_UINT64,
        PWR_NUM_ATTR_DATA_TYPES,
        /* */
        PWR_ATTR_DATA_INVALID = -1,
        PWR_ATTR_DATA_NOT_SPECIFIED = -2
} PWR_AttrDataType;
```

**PWR_AttrAccessError**

```
typedef struct {
        PWR_Obj obj;
        PWR_AttrName name;
        int error;
} PWR_AttrAccessError;
```

## 3.6 Metadata Relevant Type Definitions

The PWR_MetaName type is required to be implemented. The type may be extended by the implementor and the additional capabilities may be exposed using an implementation specified context type (see section 3.3). If new PWR_MetaName items are added, it is required that the metadata name be specified and commented as shown in the PWR_MetaName definition. Additional metadata items must be added prior to PWR_NUM_META_NAMES.

**PWR_MetaName**

```
typedef enum {
        PWR_MD_NUM = 0, /* uint64_t */
        PWR_MD_MIN, /* either uint64_t or double, depending on attribute type */
        PWR_MD_MAX, /* either uint64_t or double, depending on attribute type */
        PWR_MD_PRECISION, /* uint64_t */
        PWR_MD_ACCURACY, /* double */
        PWR_MD_UPDATE_RATE, /* double */
        PWR_MD_SAMPLE_RATE, /* double */
        PWR_MD_TIME_WINDOW, /* PWR_Time */
        PWR_MD_TS_LATENCY, /* PWR_Time */
        PWR_MD_TS_ACCURACY, /* PWR_Time */
        PWR_MD_MAX_LEN, /* uint64_t, max strlen of any returned metadata string. */
        PWR_MD_NAME_LEN, /* uint64_t, max strlen of PWR_MD_NAME */
        PWR_MD_NAME, /* char *, C-style NULL-terminated ASCII string */
        PWR_MD_DESC_LEN, /* uint64_t, max strlen of PWR_MD_DESC */
        PWR_MD_DESC, /* char *, C-style NULL-terminated ASCII string */
        PWR_MD_VALUE_LEN, /* uint64_t, max strlen returned by PWR_MetaValueAtIndex */
        PWR_MD_VENDOR_INFO_LEN, /* uint64_t, max strlen of PWR_MD_VENDOR_INFO */
        PWR_MD_VENDOR_INFO, /* char *, C-style NULL-terminated ASCII string */
        PWR_MD_MEASURE_METHOD, /* uint64_t, 0/1 depending on real/model mesurement */
        PWR_NUM_META_NAMES,
        /* */
        PWR_MD_INVALID = -1,
        PWR_MD_NOT_SPECIFIED = -2
} PWR_MetaName;
```

## 3.7 Error Return Definitions

The following required definitions are the available error returns for the functions described in this specification. It is anticipated that this list will grow. The implementor is also free to add error returns to express conditions not currently covered in the specification and expose them using an implementation specified context type (see section 3.3). The range -127 through 128 are reserved for use by the Power API specification. Positive numbers greater than zero are to be used for warnings.

```
#define PWR_RET_WARN_NOT_OPTIMIZED 1
#define PWR_RET_SUCCESS 0
#define PWR_RET_FAILURE -1
#define PWR_RET_NOT_IMPLEMENTED -2
#define PWR_RET_EMPTY -3
#define PWR_RET_INVALID -4
#define PWR_RET_LENGTH -5
#define PWR_RET_NO_ATTRIB -6
#define PWR_RET_NO_META -7
#define PWR_RET_READ_ONLY -8
#define PWR_RET_BAD_VALUE -9
#define PWR_RET_BAD_INDEX -10
#define PWR_RET_OP_NOT_ATTEMPTED -11
#define PWR_RET_NO_PERM -12
#define PWR_RET_OUT_OF_RANGE -13
```

## 3.8 Time Related Definitions

PWR_Time is defined as a 64-bit value used to hold timestamps in nanoseconds for a wide range of functionality. For those timestamps that are to be used in relation to an epoch, midnight January 1st, 1970 will be considered the beginning of the epoch. This will provide for hundreds of years to be expressed from the epoch point, which is sufficient for the purposes of the Power API. PWR_Time is also used for other structures designed to record time values (PWR_TimePeriod, page 30 for example). PWR_TIME_UNINIT is used as an indicator that the time value has not been initialized. This is intended to allow the implementation to make decisions on how a function is being used based on whether a time value has been specified or not (for example, the Statistics functions in section 4.6). PWR_TIME_UNKNOWN is an output, which indicates that the time of an event was not recorded. For example, a maximum value for an attribute could be known for a given time period, but the instant at which the maximum occurred is unknown. The PWR_TimePeriod type allows for three timestamps, start, stop and instant. Instant is available to indicate when a statistically significant event occurred within the window delineated by start and stop. For example, if the user requests the PWR_ATTR_STAT_MAX statistic for PWR_ATTR_POWER, the start and stop times will indicate the window of time over which the maximum value was calculated. The instant would indicate the instant in time the maximum value occurred. Defining PWR_Time, PWR_TIME_UNINIT, PWR_TIME_UNKNOWN, and PWR_TimePeriod as specified is required.

```
typedef uint64_t PWR_Time;
#define PWR_TIME_UNINIT 0
#define PWR_TIME_UNKNOWN 0
```

### PWR_TimePeriod

```
typedef struct {
        PWR_Time start;
        PWR_Time stop;
        PWR_Time instant;
} PWR_TimePeriod;
```

## 3.9 Statistics Relevant Type Definitions

The PWR_AttrStat type includes the list of currently defined statistics potentially available to the user of an implementation. Potentially, because this feature requires either direct device or software support. Statistics are generated on a per-attribute basis (see PWR_AttrName on page 26). The statistics type definitions are required to be implemented and are used with the statistics functions (see section 4.6).

### PWR_AttrStat

```
typedef enum {
        PWR_ATTR_STAT_MIN = 0,
        PWR_ATTR_STAT_MAX,
        PWR_ATTR_STAT_AVG,
        PWR_ATTR_STAT_STDEV,
        PWR_ATTR_STAT_CV,
        PWR_NUM_ATTR_STATS,
        /* */
        PWR_ATTR_STAT_INVALID = -1,
        PWR_ATTR_STAT_NOT_SPECIFIED = -2
} PWR_AttrStat;
```

### PWR_ID

```
typedef enum {
        PWR_ID_USER = 0,
        PWR_ID_JOB,
        PWR_ID_RUN,
        PWR_NUM_IDS,
        /* */
        PWR_ID_INVALID = -1,
        PWR_ID_NOT_SPECIFIED = -2
} PWR_ID;
```

## 3.10 OS Hardware Interface Type Definitions

The following definitions are used in the Operating system to Hardware interface described in section 6.1. Each definition will be described below along with its specification. All of the definitions in this section are required, even if the corresponding OS/HW functions are not implemented.

#### PWR_OperState

The `PWR_OperState` type is used to describe the state being requested by OS to Hardware interface functions that require power/performance state information such as P-State and C-State information. Both `c_state_num` and `p_state_num` must be provided.

```
typedef struct {
       uint64_t c_state_num;
       uint64_t p_state_num;
} PWR_OperState;
```

## 3.11 Application OS Interface Type Definitions

The following definitions are primarily used in the Application to Operating system interface described in section 6.3. Each definition will be described below along with its specification. All of the definitions in this section are required, even if the corresponding App/OS functions are not implemented.

#### PWR_RegionHint

The `PWR_RegionHint` type is an abstraction intended to allow the application to communicate power and performance significant information to the operating system. It is used in conjunction with `PWR_RegionIntensity` to describe the type and extent of the behavior described for a given execution region. This information can then be used to *tune* components, with the intent being a more power/performance efficient use of the components results. For example, if an application is going into a serial region, the performance of the application may benefit from the core running the serial portion of the code at a higher frequency, thereby completing that serial portion faster. Since the application is in a serial portion, the implementation may determine that the remaining cores may be put into a more power efficient state (a sleep state for example), thus possibly resulting in both a performance increase and a decrease in the amount of power/energy the application uses. Regions may be specified as `PWR_REGION_DEFAULT` to indicate that the application is no longer providing a hint as to the region characteristics of currently executing code.

```
typedef enum {
        PWR_REGION_DEFAULT = 0,
        PWR_REGION_SERIAL,
        PWR_REGION_PARALLEL,
        PWR_REGION_COMPUTE,
        PWR_REGION_COMMUNICATE,
        PWR_REGION_IO,
        PWR_REGION_MEM_BOUND,
        PWR_NUM_REGION_HINTS,
        /* */
        PWR_REGION_INVALID = -1,
        PWR_REGION_NOT_SPECIFIED = -2
} PWR_RegionHint;
```

### PWR_RegionIntensity

The `PWR_RegionIntensity` type is an abstraction of a given level of intensity for a PWR_RegionHint. It provides five levels of intensity as well as `PWR_Region_INT_NONE`, which can be used in the case where the intensity is not known, is not applicable, or in cases where the operating system or runtime may be better equipped to determine the intensity of a given code region.

```
typedef enum {
        PWR_REGION_INT_HIGHEST = 0,
        PWR_REGION_INT_HIGH,
        PWR_REGION_INT_MEDIUM,
        PWR_REGION_INT_LOW,
        PWR_REGION_INT_LOWEST,
        PWR_REGION_INT_NONE,
        PWR_NUM_REGION_INTENSITIES,
        /* */
        PWR_REGION_INT_INVALID = -1,
        PWR_REGION_INT_NOT_SPECIFIED = -2
} PWR_RegionIntensity;
```

### PWR_SleepState

The `PWR_SleepState` type is a high level abstraction of the different sleep state levels that may be provided on a given system. The sleep levels are translated into the appropriate hardware level constructs by lower layers of the PowerAPI.

```
typedef enum {
        PWR_SLEEP_NO = 0,
        PWR_SLEEP_SHALLOW,
        PWR_SLEEP_MEDIUM,
        PWR_SLEEP_DEEP,
        PWR_SLEEP_DEEPEST,
        PWR_NUM_SLEEP_STATES,
        /* */
        PWR_SLEEP_INVALID = -1,
        PWR_SLEEP_NOT_SPECIFIED = -2
} PWR_SleepState;
```

### PWR_PerfState

The `PWR_PerfState` type is an abstraction meant to describe the different possible performance states in which hardware may be placed.

```
typedef enum {
        PWR_PERF_FASTEST = 0,
        PWR_PERF_FAST,
        PWR_PERF_MEDIUM,
        PWR_PERF_SLOW,
        PWR_PERF_SLOWEST,
        PWR_NUM_PERF_STATES,
        /* */
        PWR_PERF_INVALID = -1,
        PWR_PERF_NOT_SPECIFIED = -2
} PWR_PerfState;
```

# Chapter 4

# Core (Common) Interface Functions

Core, or so called Common, interface functions are functions that can be used, at least in par, by most of the interfaces described in the Power API specification. Core functions include the following areas:

- **Initialization**, required to use any of the functionality described in this specification,
- **Navigation** functions allow the user to traverse the system description and discover information about the underlying platform,
- **Group** functions, primarily a convenience abstraction,
- **Attribute** functions expose measurement and control functionality,
- **Metadata** functions allow the user to access additional information about objects and attributes (often device or instrumentation specific information),
- **Statistics** functions are used to generate statistical information based on fundamental attribute information (measurements),

and other functionality that is common across a number of interfaces.

## 4.1 Initialization

Initialization using `PWR_CntxtInit` is required to use any of the functionality documented in this specification. The user supplies the type of the context requested and their role. Currently, the specification's only required context type is `PWR_CNTXT_DEFAULT`. The context type is intended to be one way in which the implementor can distinguish their implementation from the standard specification and other implementations (see section 3.3). The user must also supply their role (see page 25 for the `PWR_Role` definition). One purpose of specifying the role is to convey what type of user they intend to be, and therefore, how they would like to interact with or how the underlying implementation manages the privileges granted to the user/role combination. A system administrator (`PWR_ROLE_ADMIN`) will desire and require different capabilities, privileges and level of abstraction than the application user (`PWR_ROLE_APP`), for example.

The user also has the opportunity to specify a name that will be associated with the context. This *feature* is anticipated to be useful in supporting advanced functionality. Initialization returns a context to the user. The context contains the user's view of the system, dependent on what type

of context was requested, the user's role and implementation specifics. The system description that the user is exposed to must conform to the rules outlined in the specification (see sections 2.2 and 2.4). The context should be destroyed (cleaned up) by using the PWR_CntxtDestroy function when no longer needed.

### Function Prototype PWR_CntxtInit()

The PWR_CntxtInit function is required to be called before using any other Power API function. The context returned is passed to other Power API functions either explicitly as an argument or implicitly through an argument associated with the context.

```
PWR_Cntxt PWR_CntxtInit( PWR_CntxtType type,
                         PWR_Role role,
                         const char* name );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_CntxtType type | Input | The requested context type (see page 24). |
| PWR_Role role | Input | The role of the user (see page 25). |
| const char* name | Input | User specified string name to be associated with the context. |

| Return Code(s) | Description |
|---|---|
| PWR_Cntxt | A valid context is returned upon SUCCESS |
| NULL | A null pointer is returned upon FAILURE |

### Function Prototype for PWR_CntxtDestroy()

The PWR_CntxtDestroy function is used to destroy (clean up) the context obtained with PWR_CntxtInit.

```
int PWR_CntxtDestroy( PWR_Cntxt context );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Cntxt context | Input | The context obtained using PWR_CntxtInit the user wishes to destroy. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

## 4.2 Hierarchy Navigation Functions

Hierarchy navigation (also called discovery) is accomplished using attributes (EntryPoint, Type, Parent and Children) that are implicit to every object in the system description whether defined in the specification or added by the implementor. Navigation is accomplished using these attributes, through the associated function calls, within the context made available to the user upon initialization. After initialization the first call will generally be PWR_CntxtGetEntryPoint to determine the user's current position in the system hierarchy provided within the user's context. Depending on the user, the interface and the role, the context could contain a view of the entire system description or a subset of the system description. Navigating through the hierarchy is accomplished with PWR_ObjGetParent to navigate up and PWR_ObjGetChildren to navigate down. To understand what kind of object was returned with either of these calls the user can utilize PWR_ObjGetType call. The name of the object can be discovered using the PWR_ObjGetName function and if the user has a name, the associated object can be discovered using the PWR_CntxtGetObjByName function.

The Power API does not provide an explicit "Free Object" interface. Specifically, objects returned by Power API interfaces do not need to be later freed or released explicitly. This design choice was made in order to keep usage of the Power API as simple as possible, with the potential cost of an increased burden on the Power API implementor to limit implementation-internal memory usage.

#### Function Prototype for PWR_CntxtGetEntryPoint()

The PWR_CntxtGetEntryPoint call is typically used immediately following initialization. Given the context obtained with PWR_CntxtInit, PWR_CntxtGetEntryPoint returns the users current location, the object the user is currently pointing to in the system description contained in the context.

```
PWR_Obj PWR_CntxtGetEntryPoint( PWR_Cntxt context );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Cntxt context | Input | The user's context. |

| Return Code(s) | Description |
|---|---|
| PWR_Obj | Returns the object representing the user's current position in the system hierarchy upon SUCCESS. |
| NULL | A null pointer is returned upon FAILURE |

### Function Prototype for PWR_ObjGetType()

The PWR_ObjGetType function returns the type of the object specified. See page 25 for valid object types.

```
PWR_ObjType PWR_ObjGetType( PWR_Obj object );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj object | Input | The object that the user wishes to determine the type of. |

| Return Code(s) | Description |
|---|---|
| PWR_ObjType | Type of the specified object upon SUCCESS |
| PWR_OBJ_INVALID | Upon FAILURE |

### Function Prototype for PWR_ObjGetName()

The PWR_ObjGetName function returns the name of the object specified. See page 40 to get the object based on the unique name using PWR_CntxtGetObjByName.

```
const char* PWR_ObjGetName( PWR_Obj object );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj object | Input | The object that the user wishes to determine the name of. |

| Return Code(s) | Description |
|---|---|
| const char* | Name of the specified object upon SUCCESS. The returned pointer is marked const and should not be modified by the caller. This pointer is only valid while the object's parent context is valid. When the parent context is destroyed the pointer becomes invalid. |
| NULL | Upon FAILURE |

### Function Prototype for PWR_ObjGetParent()

The PWR_ObjGetParent function returns the object immediately above the specified object in the system description available to the user through the current context.

```
PWR_Obj PWR_ObjGetParent( PWR_Obj object );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj object | Input | The object that the user wishes to determine the parent of. |

| Return Code(s) | Description |
|---|---|
| PWR_Obj | The parent object of the specified object upon SUCCESS |
| NULL | Upon FAILURE |

### Function Prototype for PWR_ObjGetChildren()

The PWR_ObjGetChildren function returns the child or children of the specified object. A group (PWR_Grp) is returned regardless of whether the object has one or many children. Children are immediate siblings of the specified parent object. The user is responsible for destroying the group when no longer needed (see PWR_GrpDestroy 41).

```
PWR_Grp PWR_ObjGetChildren( PWR_Obj object );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj object | Input | The object that the user wishes to determine the children of. |

| Return Code(s) | Description |
|---|---|
| PWR_Grp | Group of objects (possibly a group of one object) containing the children of the specified object upon SUCCESS |
| NULL | Upon FAILURE |

**Function Prototype for PWR_CntxtGetObjByName()**

The PWR_CntxtGetObjByName function returns the object given the context and unique object name. See page 38 to get the name of a specified object using PWR_ObjGetName.

```
PWR_Obj PWR_CntxtGetObjByName( PWR_Cntxt context,
                               const char* name );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Cntxt context | Input | The context containing the object that the user wishes to retrieve given its unique name. Note, the object may be present in the system but not available to the user through the current context. |
| const char * name | Input | The unique name of the object that the user wishes to retrieve. |

| Return Code(s) | Description |
|---|---|
| PWR_Obj | Upon SUCCESS, the object associated with the unique specified name |
| NULL | Upon FAILURE |

## 4.3 Group Functions

Group functions are provided as a convenience in situations, for example, where an operation, or operations are required to be executed on multiple objects. Rather than executing the same operation multiple times, once for each object, some operations provide a group variant to streamline this type of functionality. Groups can be dynamically created (PWR_GrpCreate) when needed and can exist for short periods of time and destroyed with PWR_GrpDestroy, or exist for the duration of the users context.

### Function Prototype for PWR_GrpCreate()

The PWR_GrpCreate function is used to create a new group which will be associated with and unique to the users context.

```
PWR_Grp PWR_GrpCreate( PWR_Cntxt context );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Cntxt context | Input | The user's context that the group, when created, will be associated with. |

| Return Code(s) | Description |
|---|---|
| PWR_Grp | Upon SUCCESS (empty group) |
| NULL | Upon FAILURE |

### Function Prototype for PWR_GrpDestroy()

The PWR_GrpDestroy function is used to destroy (clean up) a group created by a user.

```
int PWR_GrpDestroy( PWR_Grp group );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group | Input | The group that the user is acting on. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

### Function Prototype for PWR_GrpDuplicate()

The PWR_GrpDuplicate function is used to duplicate an existing group.

```
PWR_Grp PWR_GrpDuplicate( PWR_Grp group );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group | Input | The group that the user is acting on. |

| Return Code(s) | Description |
|---|---|
| PWR_Grp | Upon SUCCESS (duplicated group) |
| NULL | Upon FAILURE |

### Function Prototype for PWR_GrpUnion()

The PWR_GrpUnion function is used to create a group that is the union (∪) of two specified groups.

```
PWR_Grp PWR_GrpUnion( PWR_Grp group1, PWR_Grp group2);
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group1 | Input | The first of the two groups used in the union (∪) operation. |
| PWR_Grp group2 | Input | The second of the two groups used in the union (∪) operation. |

| Return Code(s) | Description |
|---|---|
| PWR_Grp | Upon SUCCESS (union of groups input) |
| NULL | Upon FAILURE |

### Function Prototype for PWR_GrpIntersection()

The PWR_GrpIntersection function is used to create a group that is the Intersection (∩) of two specified groups.

```
PWR_Grp PWR_GrpIntersection( PWR_Grp group1, PWR_Grp group2);
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group1 | Input | The first of the two groups used in the Intersection (∩) operation. |
| PWR_Grp group2 | Input | The second of the two groups used in the intersection (∩) operation. |

| Return Code(s) | Description |
|---|---|
| PWR_Grp | Upon SUCCESS (Intersection of groups input) |
| NULL | Upon FAILURE |

### Function Prototype for PWR_GrpDifference()

The PWR_GrpDifference function is used to create a group that is the Difference (\) of two specified groups.

```
PWR_Grp PWR_GrpDifference( PWR_Grp group1, PWR_Grp group2);
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group1 | Input | The first of the two groups used in the Difference (\) operation. |
| PWR_Grp group2 | Input | The second of the two groups used in the Difference (\) operation. |

| Return Code(s) | Description |
|---|---|
| PWR_Grp | Upon SUCCESS (Difference of groups input) |
| NULL | Upon FAILURE |

### Function Prototype for PWR_GrpGetNumObjs()

The PWR_GrpGetNumObjs function is used to get the number of objects contained in the specified group.

```
int PWR_GrpGetNumObjs( PWR_Grp group );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group | Input | The group that the user is acting on. |

| Return Code(s) | Description |
|---|---|
| int | Upon SUCCESS, the number of objects contained in the specified group. |
| PWR_RET_FAILURE | Upon FAILURE |

### Function Prototype for PWR_GrpGetObjByIndx()

The PWR_GrpGetObjByIndx is used to get the object from the specified group at the specified index.

```
PWR_Obj PWR_GrpGetObjByIndx( PWR_Grp group,
                             int index );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group | Input | The group that the user is acting on. |
| int index | Input | The index within the specified group of the desired object. |

| Return Code(s) | Description |
|---|---|
| PWR_Obj | Upon SUCCESS, the object at the index within the specified group. |
| NULL | Upon FAILURE |

### Function Prototype for PWR_GrpAddObj()

The PWR_GrpAddObj function is used to add a specified object to a specified group.

```
int PWR_GrpAddObj( PWR_Grp group,
                   PWR_Obj object );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group | Input | The group that the user is acting on. |
| PWR_Obj object | Input | The object to be added to the specified group. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon SUCCESS |

### Function Prototype for PWR_GrpRemoveObj()

The PWR_GrpRemoveObj function is used to remove a specified object from a specified group.

```
int PWR_GrpRemoveObj( PWR_Grp group,
                      PWR_Obj object );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group | Input | The group that the user is acting on. |
| PWR_Obj object | Input | The object to be removed from the specified group. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

### Function Prototype for PWR_CntxtGetGrpByName()

The PWR_CntxtGetGrpByName function returns the object given the context and unique group name. This function is included to allow the user to make use of groups that are provided with the initial context by the implementation. The list of valid group names should be provided by the vendor in their documentation. Due to the defined group names being vendor specific, use of this function should be considered non-portable.

```
PWR_Grp PWR_CntxtGetGrpByName( PWR_Cntxt context,
                               const char* name );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Cntxt context | Input | The context containing the group that the user wishes to retrieve given its unique name. |
| const char * name | Input | The unique name of the group that the user wishes to retrieve. |

| Return Code(s) | Description |
|---|---|
| PWR_Grp | Upon SUCCESS, the group associated with the unique specified name. |
| NULL | Upon FAILURE |

## 4.4 Attribute Functions

The Attribute functions make up the foundation of the Power API specification, providing measurement (get) and control (set) interfaces for a wide range of power and energy related functionality. Get and set interfaces are provided for single attribute/single object, multiple attribute/single object, single attribute/multiple objects (group) and multiple attributes/multiple objects (group). In each case the user specifies the attribute or attributes to get or set. The valid attribute names are defined in the PWR_AttrName structure (see page 26). A complete list of all the valid attributes and their meanings can be found in table 4.1, section 4.8. The timestamp is a critical part of the get (measurement) interface for power and energy related information. It is very important that the timestamp returned (PWR_Time) be an accurate representation of when the value returned was measured to the best possible temporal accuracy, not when the function was called. It is required by the specification that the value returned is the value that was measured as close as possible to when the get function was called. The quality of the measurement and timestamp are device and implementation dependent. Information about each attribute can be obtained through the metadata interface, described in section 4.5.

#### Function Prototype for PWR_ObjAttrGetValue()

The PWR_ObjAttrGetValue function is provided to get the value of a single specified attribute (PWR_AttrName attr) from a single specified object (PWR_Obj object). The timestamp returned (PWR_Time *ts) should accurately represent when the value was measured.

```
int PWR_ObjAttrGetValue( PWR_Obj object,
                         PWR_AttrName attr,
                         void* value,
                         PWR_Time* ts );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj object | Input | The target object. |
| PWR_AttrName attr | Input | The target attribute. See section 3.5 for a list of available attributes. |
| void* value | Output | Pointer to caller-allocated storage, of 8 bytes, to hold the value read from the attribute. |
| PWR_Time* ts | Output | Pointer to caller-allocated storage to hold the timestamp of when the value was read from the attribute. Pass in NULL if the timestamp is not needed. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NOT_IMPLEMENTED | The requested attribute is not supported for the target object. |

### Function Prototype for PWR_ObjAttrSetValue()

The PWR_ObjAttrSetValue function is provided to set the value of a single specified attribute (PWR_AttrName attr) of a single specified object (PWR_Obj object).

```
int PWR_ObjAttrSetValue( PWR_Obj object,
                         PWR_AttrName attr,
                         const void* value );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj object | Input | The target object. |
| PWR_AttrName attr | Input | The target attribute. See section 3.5 for a list of available attributes. |
| const void* value | Input | Pointer to the 8 byte value to write to the attribute. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NOT_IMPLEMENTED | The requested attribute is not supported for the target object. |
| PWR_RET_BAD_VALUE | The value was not appropriate for the target attribute. |
| PWR_RET_OUT_OF_RANGE | The value was out of range for the target attribute. |

### Function Prototype for PWR_StatusCreate()

The PWR_StatusCreate function is provided to create the PWR_Status structure that will be used in functions that perform multiple operations and potentially return individual statuses for each operation. It is up to the implementation to create the appropriate amount of storage for the PWR_Status structure based on the implementation and the number of statuses that will be held. For example see PWR_ObjAttrGetValues on page 50. Note, PWR_Status is an opaque handle, its backing definition is determined by the implementor (see 3.1). It is intended that the implementation only allocate space for failed operations. Errors are read from the PWR_Status by popping them off the structure which requires the structure to only be as large as the number of error returns require.

```
PWR_Status PWR_StatusCreate( );
```

| Return Code(s) | Description |
|---|---|
| PWR_Status | Upon SUCCESS |
| NULL | Upon FAILURE |

### Function Prototype for PWR_StatusDestroy()

The PWR_StatusDestroy function is provided to destroy the PWR_Status structure created using PWR_StatusCreate (see page 48. Note, PWR_Status is an opaque handle, its backing definition is determined by the implementor (see 3.1).

```
int PWR_StatusDestroy( PWR_Status status );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Status status | Input | The PWR_Status structure the user wishes to destroy. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

### Function Prototype for PWR_StatusPopError()

The PWR_StatusPopError function is provided to iterate through the PWR_Status structure created using PWR_StatusCreate (see page 48) and populated using any of the function calls that leverage this structure. Using this method allows the PWR_Status structure to only grow as large as necessary storing only error returns. Note, PWR_Status is an opaque handle, its backing definition is determined by the implementor (see 3.1).

```
int PWR_StatusPopError( PWR_Status status,
                        PWR_AttrAccessError* error );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Status status | Input | The PWR_Status structure the user wishes to examine (iterate over). |
| PWR_AttrAccessError* error | Output | Pointer to a PWR_AttrAccessError structure (see page 27) to hold the status that is popped from the PWR_Status structure. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_EMPTY | Returned when all errors have been popped |
| PWR_RET_FAILURE | Upon FAILURE |

### Function Prototype for PWR_StatusClear()

The PWR_StatusClear function is provided to clear a previously used PWR_Status structure created using PWR_StatusCreate, (see page 48) basically allowing reuse of the same structure if multiple calls are executed and examined in sequence. Note, PWR_Status is an opaque handle, its backing definition is determined by the implementor (see 3.1).

```
int PWR_StatusClear( PWR_Status status )
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Status status | Input | The PWR_Status structure the user wishes to clear (reuse). |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

### Function Prototype for PWR_ObjAttrGetValues()

The PWR_ObjAttrGetValues function is provided to get the value of multiple specified attributes listed in the PWR_AttrName attrs[] array from a single specified object – **get multiple attribute values from a single object**. The timestamps returned in the PWR_Time ts[] array should accurately represent, and correspond sequentially, with the time each value returned was measured. If the function fails for one or more attributes, the PWR_Status status structure returned can be examined for additional information regarding the failure using PWR_StatusPopError (see page 49).

```
int PWR_ObjAttrGetValues( PWR_Obj object,
                          int count,
                          const PWR_AttrName attrs[],
                          void* values,
                          PWR_Time ts[],
                          PWR_Status status );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj object | Input | The target object. |
| int count | Input | The number of elements in the attrs[], *values, and ts[] arrays. |
| const PWR_AttrName attrs[] | Input | The array of target attributes to read. See section 3.5 for a list of available attributes. |
| void* values | Output | The array of values read, one value for each target attribute. This should point to caller-allocated storage of at least (count * 8) bytes. Upon success, the value read for attribute attrs[i] will be located at address (values+(i*8)). |
| PWR_Time ts[] | Output | The array of timestamps, one timestamp for each value read. This should point to caller-allocated storage of at least (count*sizeof(PWR_Time)). Upon success, the timestamp of the value read for attrs[i] will be located at ts[i]. Pass in NULL if timestamps are not needed. |
| PWR_Status status | Output | Upon PWR_RET_FAILURE, status contains information about each failure that occurred. Pass in NULL if failure information is not needed. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS, all operations succeeded. |
| PWR_RET_FAILURE | Upon FAILURE, one or more operations failed. Examine PWR_Status status to determine the operations that failed. All other operations succeeded. |

### Function Prototype for PWR_ObjAttrSetValues()

The PWR_ObjAttrSetValues function is provided to set the value of multiple specified attributes in the (PWR_AttrName attrs[]) array of a specified object – **set multiple attribute values of a single object**. If the function fails for one or more attributes, the PWR_Status status structure returned can be examined for additional information regarding the failure using PWR_StatusPopError (see page 49).

```
int PWR_ObjAttrSetValues( PWR_Obj object,
                          int count,
                          const PWR_AttrName attrs[],
                          const void* values,
                          PWR_Status status );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj object | Input | The target object. |
| int count | Input | The number of elements in the attrs[] and *values arrays. |
| const PWR_AttrName attrs[] | Input | The array of target attributes to write. See section 3.5 for a list of available attributes. |
| const void* values | Input | The array of values to write, one value for each target attribute. The value to write to attribute attrs[i] is located at address (values+(i*8)). |
| PWR_Status status | Output | Upon PWR_RET_FAILURE, status contains information about each failure that occurred. Pass in NULL if failure information is not needed. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS, all operations succeeded. |
| PWR_RET_FAILURE | Upon FAILURE, one or more operations failed. Examine PWR_Status status to determine the operations that failed. All other operations succeeded. |

### Function Prototype for PWR_ObjAttrIsValid()

The PWR_ObjAttrIsValid function is used to determine if a specified attribute (PWR_AttrName attr) is valid for the specified object.

```
int PWR_ObjAttrIsValid( PWR_Obj object,
                        PWR_AttrName attr );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj object | Input | The object that the user is acting on. |
| PWR_AttrName attr | Input | The attribute the user wishes to confirm is valid for the specified object. See the PWR_AttrName type definition in section 3.5. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

**Function Prototype for PWR_GrpAttrGetValue()**

The PWR_GrpAttrGetValue function is provided to get the value of a single specified attribute (PWR_AttrName attr) from all the objects in a specified group (PWR_Grp group) – **get a single attribute value from multiple objects**. The timestamps returned in the PWR_Time ts[] array should accurately represent, and correspond sequentially, with the time each value returned was measured. If the function fails for one or more attributes, the PWR_Status status structure returned can be examined for additional information regarding the failure using PWR_StatusPopError (see page 49). PWR_GrpAttrGetValue will continue to attempt to gather values for the entire group, even if an error occurs for a subset of the members of that group.

```
int PWR_GrpAttrGetValue( PWR_Grp group,
                         PWR_AttrName attr,
                         void* values,
                         PWR_Time ts[],
                         PWR_Status status );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group | Input | The target group. |
| PWR_AttrName attr | Input | The target attribute to retrieve (get) from each object in the target group. See section 3.5 for a list of available attributes. |
| void* values | Output | The array of attribute values retrieved, one value for each object in the target group. This should point to caller-allocated storage of at least (PWR_GrpGetNumObjs() * 8) bytes. Upon success, the value retrieved for the object at index i within the group will be located at address (values+(i*8)). |
| PWR_Time ts[] | Output | The array of timestamps, one timestamp for each value retrieved. This should point to caller-allocated storage of at least (PWR_GrpGetNumObjs()*sizeof(PWR_Time)). Upon success, the timestamp of the value retrieved for the object at index i within the group will be located at ts[i]. Pass in NULL if timestamps are not needed. |
| PWR_Status status | Output | Upon PWR_RET_FAILURE, status contains information about each failure that occurred. Pass in NULL if failure information is not needed. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS, all operations succeeded. |
| PWR_RET_FAILURE | Upon FAILURE, one or more operations failed. Examine PWR_Status status to determine the operations that failed. All other operations succeeded. |

### Function Prototype for PWR_GrpAttrSetValue()

The PWR_GrpAttrSetValue function is provided to set the value of a single specified attribute (PWR_AttrName attr) of each object in a specified group – **set a single attribute value on multiple objects**. If the function fails for one or more attributes, the PWR_Status status structure returned can be examined for additional information regarding the failure using PWR_StatusPopError (see page 49). PWR_GrpAttrSetValue will continue to attempt to set values for the entire group, even if an error occurs for a subset of the members of that group.

```
int PWR_GrpAttrSetValue( PWR_Grp group,
                         PWR_AttrName attr,
                         const void* value,
                         PWR_Status status );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group | Input | The target group. |
| PWR_AttrName attr | Input | The target attribute to set for each object in the target group. See section 3.5 for a list of available attributes. |
| const void* value | Input | The pointer to a single 8 byte attribute value to set for each object in the target group. |
| PWR_Status status | Output | Upon PWR_RET_FAILURE, status contains information about each failure that occurred. Pass in NULL if failure information is not needed. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS, all operations succeeded. |
| PWR_RET_FAILURE | Upon FAILURE, one or more operations failed. Examine PWR_Status status to determine the operations that failed. All other operations succeeded. |

### Function Prototype for PWR_GrpAttrGetValues()

The PWR_GrpAttrGetValues function is provided to get the value of multiple specified attributes listed in the PWR_AttrName attrs[] array from each object in a specified group – **get multiple attribute values from multiple objects**. The timestamps returned in the PWR_Time ts[] array should accurately represent, and correspond sequentially, with the time each value returned was measured. If the function fails for one or more attributes, the PWR_Status status structure returned can be examined for additional information regarding the failure using PWR_StatusPopError (see page 49). PWR_GrpAttrGetValues will continue to attempt to gather values for the entire group, even if an error occurs for a subset of the members or attributes requested in the object group.

```
int PWR_GrpAttrGetValues( PWR_Grp group,
                          int count,
                          const PWR_AttrName attrs[],
                          void* values,
                          PWR_Time ts[],
                          PWR_Status status );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group | Input | The target group. |
| int count | Input | The number of elements in the attrs[] array. |
| const PWR_AttrName attrs[] | Input | The array specifying the set of target attributes to read for each object in the target group. See section 3.5 for a list of available attributes. |
| void* values | Output | The array of attribute values retrieved. This should point to caller-allocated storage of at least (PWR_GrpGetNumObjs()*count*8) bytes. Upon success, the value read for attribute attrs[i] for the object at index j within the group will be located at address (values+(j*count*8)+(i*8)). |
| PWR_Time ts[] | Output | The array of timestamps, one timestamp for each value retrieved. This should point to caller-allocated storage of at least (PWR_GrpGetNumObjs()*count*sizeof(PWR_Time)). Upon success, the timestamp of the value retrieved for attribute attrs[i] for the object at index j within the group will be located at ts[(j*count)+i]. Pass in NULL if timestamps are not needed. |
| PWR_Status status | Output | Upon PWR_RET_FAILURE, status contains information about each failure that occurred. Pass in NULL if failure information is not needed. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS, all operations succeeded. |
| PWR_RET_FAILURE | Upon FAILURE, one or more operations failed. Examine PWR_Status status to determine the operations that failed. All other operations succeeded. |

### Function Prototype for PWR_GrpAttrSetValues()

The PWR_GrpAttrSetValues function is provided to set the value of multiple specified attributes listed in the (PWR_AttrName attrs[]) array of each object in a specified group – **set multiple attribute values on multiple objects**. If the function fails for one or more attributes, the PWR_Status status structure returned can be examined for additional information regarding the fail-

ure using PWR_StatusPopError (see page 49). PWR_GrpAttrSetValues will continue to attempt to set values for the entire group and requested attributes, even if an error occurs for a subset of the members or attributes of that object group.

```
int PWR_GrpAttrSetValues( PWR_Grp group,
                          int count,
                          const PWR_AttrName attrs[],
                          const void* values,
                          PWR_Status status );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group | Input | The target group. |
| int count | Input | The number of elements in the attrs[] and *values arrays. |
| const PWR_AttrName attrs[] | Input | The array specifying the set of target attributes to set for each object in the target group. See section 3.5 for a list of available attributes. |
| const void* values | Input | The array of attribute values to set for each object in the group. The value to write to attribute attrs[i] of each object is located at address (values+(i*8)). |
| PWR_Status status | Output | Upon PWR_RET_FAILURE, status contains information about each failure that occurred. Pass in NULL if failure information is not needed. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS, all operations succeeded. |
| PWR_RET_FAILURE | Upon FAILURE, one or more operations failed. Examine PWR_Status status to determine the operations that failed. All other operations succeeded. |

## 4.5 Metadata Functions

The metadata functions provide an interface for getting more descriptive information about an object or attribute, such as estimated measurement accuracy or the list of valid values for a given attribute. This information is often useful for getting a better understanding of the meaning of objects and attributes and how to interpret the values read from attributes. While most metadata

is read-only information, some metadata is potentially configurable, such as the underlying power sampling rate used to calculate PWR_ATTR_ENERGY values.

Table 4.2 on page 30 lists the available types of metadata. Not all of the metadata items listed will be available for every object and attribute pair. The exact set is dependent on the capabilities of the underlying hardware and Power API implementation. If a requested metadata item is not available a PWR_RET_NO_ATTRIB error is returned at runtime.

The majority of metadata items will require that both an object instance and attribute name pair be specified, but a few may be defined for object instances alone. For example, the metadata strings PWR_MD_NAME, PWR_MD_DESC, and PWR_MD_VENDOR_INFO may be available for individual object instances, with no associated attribute name specified. In these cases, the attribute name requested should be set to PWR_ATTR_NOT_SPECIFIED. One important use case for these informational strings, especially the PWR_MD_VENDOR_INFO string, is for a Power API user to capture these strings with each run to record configuration and provenance information. For example, a user may chose to log the PWR_MD_VENDOR_INFO string for the top-level platform object in the output of each run.

The metadata interface consists of three functions. The PWR_ObjAttrGetMeta and PWR_ObjAttrSetMeta functions allow metadata values to be retrieved and set, respectively. The third function, PWR_MetaValueAtIndex, provides a way to enumerate through an attribute's list of available values. This is useful for attributes that have a small, well-defined set of discrete values (e.g., PWR_ATTR_PSTATE). It is expected that where a set of discrete values can be described in a logical order that the index ordering is from smallest (lowest) to largest (highest) value. The remainder of this section describes the metadata functions in more detail.

### Function Prototype for PWR_ObjAttrGetMeta()

The PWR_ObjAttrGetMeta function returns the requested metadata item for the specified object or object and attribute name pair. The caller must allocate enough storage to hold the returned metadata value and pass a pointer to the storage in the value argument. The required size can be determined by consulting the type column of Table 4.2. In the case of string metadata items (i.e., type char *), the required string length can be determined by getting the appropriate length metadata item, which is the original metadata name with the _LEN suffix added. For example, the required string length for the PWR_MD_VENDOR_INFO string can be determined by retrieving the PWR_MD_VENDOR_INFO_LEN metadata item.

```
int PWR_ObjAttrGetMeta( PWR_Obj obj,
                        PWR_AttrName attr,
                        PWR_MetaName meta,
                        void* value );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj obj | Input | The target object. |
| PWR_AttrName attr | Input | The target attribute. See the PWR_AttrName type definition in Section 3.5 for the list of possible attributes. If object-only metadata is being requested, this argument should be set to PWR_ATTR_NOT_SPECIFIED. |
| PWR_MetaName meta | Input | The target metadata item to get. See the PWR_MetaName type definition in Section 3.6 for the list of possible metadata items, with detailed descriptions provided in Table 4.2. |
| void* value | Output | Pointer to the caller allocated storage to hold the value of the requested metadata item. See Table 4.2 for type information. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NO_ATTRIB | The attribute specified is not implemented. |
| PWR_RET_NO_META | The metadata specified is not implemented. |

### Function Prototype for PWR_ObjAttrSetMeta()

The PWR_ObjAttrSetMeta function sets the specified metadata item for the target object or object and attribute name pair. The caller must pass a pointer to the new value for the specified metadata item in the value argument. The required type for the value can be determined by consulting the type column of Table 4.2. In the case of string metadata items (i.e., type char *), the maximum string length can be determined by getting the appropriate length metadata item, which is the original metadata name with the _LEN suffix added. For example, the maximum string length for the PWR_MD_VENDOR_INFO string can be determined by retrieving the PWR_MD_VENDOR_INFO_LEN metadata item.

```
int PWR_ObjAttrSetMeta( PWR_Obj obj,
                        PWR_AttrName attr,
                        PWR_MetaName meta,
                        const void* value );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj obj | Input | The target object. |
| PWR_AttrName attr | Input | The target attribute. See the PWR_AttrName type definition in Section 3.5 for the list of possible attributes. If object-only metadata is being set, this argument should be set to PWR_ATTR_NOT_SPECIFIED. |
| PWR_MetaName meta | Input | The target metadata item to set. See the PWR_MetaName type definition in Section 3.6 for the list of possible metadata items, with detailed descriptions provided in Table 4.2. |
| const void* value | Input | Pointer to the new value for the metadata item. See Table 4.2 for type information. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NO_ATTRIB | The attribute specified is not implemented. |
| PWR_RET_NO_META | The metadata specified is not implemented. |
| PWR_RET_READ_ONLY | The metadata specified is not settable. |
| PWR_RET_BAD_VALUE | The value specified is not valid. |

**Function Prototype for PWR_MetaValueAtIndex()**

The PWR_MetaValueAtIndex function allows the available values for a given attribute to be enumerated. It is assumed that the set of valid values is static and has size equal to the value returned by the PWR_MD_NUM metadata item. Once the value of PWR_MD_NUM is known, PWR_MetaValueAtIndex() can be called repeatedly with index from 0 to PWR_MD_NUM - 1 to retrieve the list of valid values for the target attribute. Each call will return the value at the specified index as well as a human-readable string representing the value in human readable format.

If an attribute is not enumerable, then PWR_MD_NUM will return 0. In general any attribute that does not have a small set of discrete valid values will return 0 when PWR_MD_NUM is requested, to indicate that the attribute is not enumerable.

```
int PWR_MetaValueAtIndex( PWR_Obj obj,
                          PWR_AttrName attr,
                          unsigned int index,
                          void* value,
                          char* value_str );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj obj | Input | The target object. |
| PWR_AttrName attr | Input | The target attribute. See the PWR_AttrName type definition in Section 3.5 for the list of possible attributes. |
| unsigned int index | Input | The index of the metadata item value to look up. The PWR_MD_NUM metadata item returns the number of possible values, indexed from 0 to PWR_MD_NUM - 1. |
| void* value | Output | Pointer to the caller allocated storage to hold the value of the requested metadata item value. See Table 4.2 for type information. The storage must be sized appropriately for the metadata value type. If the value is not required, this argument should be set to NULL. |
| char* value_str | Output | Pointer to the caller allocated storage to hold the human-readable C-style NULL-terminated ASCII string representing the metadata item value. The storage passed in must have size in bytes of at least the value returned by the PWR_MD_VALUE_LEN metadata item. If the string representation is not required this argument should be set to NULL. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NO_ATTRIB | The attribute specified is not implemented. |
| PWR_RET_BAD_INDEX | The index specified is not valid. |

## 4.6 Statistics Functions

The statistics functions provide an interface to generate statistics related to specific attributes of an object or group. The interface allows for generating statistics somewhat real-time or mining historic statistics, assuming that the necessary data is retained. The PWR_ObjCreateStat and PWR_GrpCreateStat functions are used to create a PWR_Stat pointer. In the case of either an object or group, a statistic is directly related to the attribute specified in the create call. Basically, a tuple of information is provided, an object or group, the attribute (PWR_ATTR_POWER for example,

see page 26) that the user would like the statistic for and the statistic (PWR_ATTR_STAT_AVG for example, see page 30).

If the statistics pointer is used in conjunction with the PWR_StatStart and the PWR_StatStop calls to start and stop the collection of information that will be used to generate the statistic, the start member of the PWR_TimePeriod structure must be initialized to PWR_TIME_UNINIT. This allows the underlying implementation to immediately, and quickly, determine which mode the interface is being used in, real-time or mining. Using this interface in the real-time mode assumes either hardware device support or lower layer software support for retaining the information over the window of time delineated by the start and stop calls. The user can collect statistics after calling start, and before calling stop by using PWR_StatGetValue when the statistic pointer was created with PWR_ObjCreateStat or PWR_StatGetValues when the statistic pointer was created with PWR_GrpCreateStat, notice the plural values in the group call. If stop has not been called on the statistic pointer each time the user requests statistics, the start time, if pertinent, will be the time start was called on the statistic pointer and the stop time, again if pertinent, will be when the call was made to retrieve the statistic (see page 30 for details regarding the PWR_TimePeriod structure). Once stop is called on the statistic pointer any call to retrieve the statistic will be over the window defined by the start and stop calls.

If supported, collecting historic information is supported using the same interface but without the use of PWR_StatStart or PWR_StatStop. To collect historic information, again assuming it is retained, the user creates an object or group statistics pointer (PWR_ObjCreateStat or PWR_GrpCreateStat) followed by using PWR_StatGetValue, for an object statistic, or PWR_StatGetValues for a group statistic. The important difference is that the PWR_TimePeriod structure is populated with the start and stop times delineating the historic window that the statistic requested will be calculated over. Note that the times the user requests may not be available. The interface is required to return the requested statistic calculated over as close to the requested time window as possible. The interface is additionally required to return in the PWR_TimePeriod structure the times that were actually used in calculating the statistic that was returned. If the instant member of the PWR_TimePeriod structure is not pertinent or available it should be set to PWR_TIME_UNKNOWN.

### Function Prototype for PWR_ObjCreateStat()

The PWR_ObjCreateStat function is used to create the statistics pointer that will be used for all subsequent statistics gathering for a single object.

```
PWR_Stat PWR_ObjCreateStat( PWR_Obj object,
                            PWR_AttrName name,
                            PWR_AttrStat statistic );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj object | Input | The object to act on. |
| PWR_AttrName name | Input | The attribute to act on, see the PWR_AttrName type definition in section 3.5. |
| PWR_AttrStat statistic | Input | The desired statistic for the specified attribute, see PWR_AttrStat type definition in section 3.9. |

| Return Code(s) | Description |
|---|---|
| PWR_Stat | Stat for that object, attribute statistic triple upon SUCCESS. |
| NULL | Upon FAILURE |

### Function Prototype for PWR_GrpCreateStat()

The PWR_GrpCreateStat function is used to create the statistics pointer that will be used for all subsequent statistics gathering for a group of objects.

```
PWR_Stat PWR_GrpCreateStat( PWR_Grp group,
                            PWR_AttrName name,
                            PWR_AttrStat statistic );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Grp group | Input | The group to act on. |
| PWR_AttrName name | Input | The attribute to act on, see the PWR_AttrName type definition in section 3.5. |
| PWR_AttrStat statistic | Input | The desired statistic for the specified attribute, see PWR_AttrStat type definition in section 3.9. |

| Return Code(s) | Description |
|---|---|
| PWR_Stat | Stat for that object, attribute statistic triple upon SUCCESS. |
| NULL | Upon FAILURE |

### Function Prototype for PWR_StatStart()

The PWR_StatStart function is used to indicate to a device or software layer to start the window of time that the statistic requested will be calculated over.

```
PWR_Stat PWR_StatStart( PWR_Stat statObj );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Stat statObj | Input | The statistics object to begin collecting the specified statistic for (specified in PWR_ObjCreateStat). |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

### Function Prototype for PWR_StatStop()

The PWR_StatStop function is used to indicate to a device or software layer to stop the window of time that the statistic requested will be calculated over.

```
PWR_Stat PWR_StatStop( PWR_Stat statObj );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Stat statObj | Input | The statistics object to stop collecting the specified statistic for (specified in PWR_ObjCreateStat). |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon SUCCESS |

### Function Prototype for PWR_StatClear()

The PWR_StatClear function is used to indicate to a device or software layer to clear or reset the window of time that the statistic requested will be calculated over. The clear effectively restarts the window, so there is no need to call PWR_StatStart again.

```
PWR_Stat PWR_StatClear( PWR_Stat statObj );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Stat statObj | Input | The statistics object to clear (effectively reset) for the specified statistic (specified in PWR_ObjCreateStat). |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

### Function Prototype for PWR_StatGetValue()

The PWR_StatGetValue function is used to retrieve the statistic and related time stamp information from the statistics pointer created using PWR_ObjCreateStat. Note that the PWR_StatGetValue call operates on single objects only, not groups of objects. If a single value return is desired for a group of objects, the PWR_StatGetReduce call on page 66 should be used.

```
PWR_Stat PWR_StatGetValue( PWR_Stat statObj,
                           double* value,
                           PWR_TimePeriod* statTimes );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Stat statObj | Input | The statistics object to collect the statistic for (the object, attribute stat triple is specified in PWR_ObjCreateStat). |
| double* value | Output | pointer to space (double) to store the statistic |
| PWR_TimePeriod* statTimes | Input/Output | Time structure that contains the timestamps pertinent to the specific statistic, see page 30. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

### Function Prototype for PWR_StatGetValues()

The PWR_StatGetValues function is used to retrieve the statistic and related time stamp information from the statistics pointer created using PWR_GrpCreateStat.

```
PWR_Stat PWR_StatGetValues( PWR_Stat statObj,
                            double values[],
                            PWR_TimePeriod statTimes[] );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Stat statObj | Input | The statistics object to collect the statistic for (the group, attribute stat triple is specified in PWR_GrpCreateStat). |
| double values[] | Output | Space allocated by user to hold array of values (statistics) |
| PWR_TimePeriod statTimes[] | Input/Output | Space allocated by user to hold array of time structures that contains the timestamps pertinent to each specific statistic, see page 30 |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

**Function Prototype for PWR_StatGetReduce()**

The PWR_StatGetReduce function is used to retrieve the statistic and related time stamp information from the statistics pointer created using PWR_GrpCreateStat and perform a reduction operation on the result. The type of reduction performed is determined by the statistic dictated for the PWR_Stat when it was created.

The value returned from a call to PWR_StatGetReduce will be equivalent to the statistic that would be calculated by calling PWR_GrpAttrGetValue and performing the operation on the returned set of values (one per group member) for the chosen attribute. PWR_StatGetReduce is provided such that optimizations that may be possible when calculating a given statistic can be utilized. An example of such an operation would be calculating an average, where gathering the values is done through a tree topology overlay network, where averages can be calculated at each parent of multiple children in the tree. Note that the implementation of PWR_StatGetReduce can be done in its more simplistic form by calling PWR_GrpAttrGetValue and performing the required operation on the returned set of values to return the requested reduction operation.

For certain reduction operations, some elements of the time stamp (PWR_TimePeriod) may not be valid output. For example, in the case of a averaging reduction, an associated "instant" time stamp is not a useful value. For Min and Max operations, the "instant" time stamp is useful and will represent the time at which the maximum or minimum was observed. In all cases the start and stop time stamps in the PWR_TimePeriod will represent the time window over which the the value was calculated.

```
int PWR_StatGetReduce( PWR_Stat statObj,
                       double* value,
                       PWR_TimePeriod* statTimes);
```

| Argument(s) | Input and/or Output Input | Description |
|---|---|---|
| PWR_Stat statObj | Input | The statistics object to collect the statistic for (the object group, attribute stat triple is specified in PWR_GrpCreateStat). |
| double* value | Output | pointer to space (double) to store the statistic |
| PWR_TimePeriod* statTimes | Input/Output | Time structure that contains the timestamps pertinent to the specific statistic, see page 30. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

**Function Prototype for PWR_StatDestroy()**

The PWR_StatDestroy function is used to destroy (clean up) the statistics pointer created using PWR_ObjCreateStat or PWR_GrpCreateStat.

```
int PWR_StatDestroy( PWR_Stat statObj );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Stat statObj | Input | The statistics object to destroy (clean up) |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |

## 4.7 Version Functions

The PWR_GetMajorVersion and PWR_GetMinorVersion functions are used to get the major and minor portions of the specification version supported by the implementation. Users can make decisions regarding available functionality based on the version number supported.

**Function Prototype for PWR_GetMajorVersion()**

The PWR_GetMajorVersion function is used to get the major version number portion of the version number of the specification supported by the implementation.

```
int PWR_GetMajorVersion( );
```

| **Return Code(s)** | **Description** |
|---|---|
| int | Upon SUCCESS, integer representation of major portion of version number |
| PWR_RET_FAILURE | Upon FAILURE |

**Function Prototype for PWR_GetMinorVersion()**

The PWR_GetMinorVersion function is used to get the minor version portion of the version number of the specification supported by the implementation.

```
int PWR_GetMinorVersion( );
```

| **Return Code(s)** | **Description** |
|---|---|
| int | Upon SUCCESS, integer representation of minor portion of version number |
| PWR_RET_FAILURE | Upon FAILURE |

## 4.8 Big List of Attributes

The following is the master list of Attributes available to the user. The attributes valid for specific interfaces are listed in the appropriate section in Chapter 6.

Table 4.1: Complete List of All Supported Attributes

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_PSTATE | Set/Get | uint64_t | The current P-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE | Set/Get | uint64_t | The current C-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE_LIMIT | Set/Get | uint64_t | The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_SSTATE | Set/Get | uint64_t | The current S-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CURRENT | Get | double | Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_VOLTAGE | Get | double | Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER | Get | double | Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER_LIMIT_MIN | Set/Get | double | Minimum power limit (floor, lower bound) for the specified object in watts. |
| PWR_ATTR_POWER_LIMIT_MAX | Set/Get | double | Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts. |
| PWR_ATTR_FREQ | Set/Get | double | The current operating frequency value for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MIN | Set/Get | double | Minimum operating frequency limit for the specified object in Hz (cycles per second). |

Continued on next page

Table 4.1 – continued from previous page

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_FREQ_LIMIT_MAX | Set/Get | double | Maximum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_ENERGY | Get | double | The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value. |
| PWR_ATTR_TEMP | Get | double | The current temperature value for the specified object in degrees Celsius. |
| PWR_ATTR_OS_ID | Get | uint64_t | The operating system ID that corresponds to the object. For example, a runtime system may need to figure out which Power API PWR_OBJ_CORE objects correspond to the cores that it is controlling. This attribute provides a linkage between Power API objects and operating system IDs. |
| PWR_ATTR_THROTTLED_TIME | Get | uint64_t | The cumulative time in nanoseconds that the specified object's performance was purposefully slowed in order to meet some constraint, such as a power cap. |
| PWR_ATTR_THROTTLED_COUNT | Get | uint64_t | The cumulative count of the number of times that the specified object's performance was purposefully slowed in order to meet some constraint, such as a power cap. |

## 4.9 Big List of Metadata

Table 4.2: Complete List of All Metadata Names

| Metadata | Set and/or Get | Type (SaA = Same as Attribute) | Description |
|---|---|---|---|
| PWR_MD_NUM | Get | uint64_t | Number of values supported. This is only relevant for attributes with a discrete set of values (e.g., PWR_ATTR_PSTATE). Other attributes return 0. |
| PWR_MD_MIN | Get | SaA | Minimum value supported. |
| PWR_MD_MAX | Get | SaA | Maximum value supported. |
| PWR_MD_PRECISION | Get | uint64_t | Number of significant digits in values. |
| PWR_MD_ACCURACY | Get | double | Estimated percent error +/- of measured vs. actual values. |
| PWR_MD_UPDATE_RATE | Set/Get | double | Rate values become visible to user, in updates per second. Getting or setting a value at a rate higher than this is not useful. |
| PWR_MD_SAMPLE_RATE | Set/Get | double | Rate of underlying sampling, in samples per second. This is only relevant for values derived over time (e.g., PWR_ATTR_ENERGY). |
| PWR_MD_TIME_WINDOW | Set/Get | PWR_Time | The time window used to calculate the value returned or relevant to an attribute. For example, the "instantaneous" PWR_ATTR_POWER values reported may actually be averaged over a short time window. Power caps are also enforced with respect to a target time window. |
| PWR_MD_TS_LATENCY | Get | PWR_Time | Estimate of the time required to get or set an attribute. This is useful to estimate completion time for an operation *a priori*. A value of zero should be returned when the get/set is instantaneous. |
| PWR_MD_TS_ACCURACY | Get | PWR_Time | Estimated accuracy of returned timestamps, represented as +/- the PWR_Time value returned. |

Continued on next page

Table 4.2 – continued from previous page

| Metadata | Set and/or Get | Type (SaA = Same as Attribute) | Description |
|---|---|---|---|
| PWR_MD_MAX_LEN | Get | uint64_t | The maximum string length that will be returned by the metadata interface. All other string lengths (metadata items ending in "_LEN") will be less than or equal to this value. The value of PWR_MD_MAX_LEN will be less than or equal to PWR_MAX_STRING_LEN. |
| PWR_MD_NAME_LEN | Get | uint64_t | Length of the attribute name string, in bytes. This is the buffer length needed to store the string returned when PWR_MD_NAME is requested. |
| PWR_MD_NAME | Get | char * | Attribute name string. This is a C-style NULL-terminated ASCII string. This provides a human readable name for the attribute. The string length is given by PWR_MD_NAME_LEN. |
| PWR_MD_DESC_LEN | Get | uint64_t | Length of the attribute description string, in bytes. This is the buffer length needed to store the string returned when PWR_MD_DESC is requested. |
| PWR_MD_DESC | Get | char * | Attribute description string. This is a C-style NULL-terminated ASCII string. This provides a human readable description of the attribute that is more descriptive than the attribute's name alone. The string length is given by PWR_MD_DESC_LEN. |
| PWR_MD_VALUE_LEN | Get | uint64_t | Maximum length of the value strings returned by PWR_MetaValueAtIndex. This can be used to discover the buffer size that needs to be passed to PWR_MetaValueAtIndex via the value_str argument. |
| PWR_MD_VENDOR_INFO_LEN | Get | uint64_t | Length of the vendor information string, in bytes. This is the buffer length needed to store the string returned when PWR_MD_VENDOR_INFO is requested. |

Continued on next page

Table 4.2 – continued from previous page

| Metadata | Set and/or Get | Type (SaA = Same as Attribute) | Description |
|---|---|---|---|
| PWR_MD_VENDOR_INFO | Get | char * | Vendor provided information string. This is a C-style NULL-terminated ASCII string. This may be used to convey part numbers, configuration, or other non-standard information. The string length is given by PWR_MD_VENDOR_INFO_LEN. |
| PWR_MD_MEASURE_METHOD | Get | uint64_t | Denotes the measurement method: an actual measurement (returned value = 0) or a model based estimate (return value = 1). Other values $> 1$ may be used to denote multiple vendor specific models in the situation where multiple models may exist. |

# Chapter 5

# High-Level (Common) Functions

This chapter includes specifications for High-Level functions that are common for more than one of the Role/System pair interfaces specified in chapter 6. The implementation may choose to selectively provide implementations for these functions, but all should be stubbed out or available. If an implementation is not provided the function should simply return `PWR_RET_NOT_IMPLEMENTED`.

## 5.1 Report Functions

Report functions are intended to provide a number of Role/System pairs with the ability to produce a range of reports. These particular functions target historic data, typically data that has been recorded in logs or some type of database. These functions are considered High-Level and abstract the object and group concepts found in the Core functions. Information is requested based on higher level concepts such as job, application or user ID. These functions require the user to provide a context which is used for determining whether the calling user can access the requested data.

#### Function Prototype PWR_GetReportByID()

The `PWR_GetReportByID` function is provided to allow the collection of statistics information based on the ID types defined in `PWR_ID` in Section 3.9. A `PWR_ID` type must be supplied with `char*` pointer pointing to a valid ID for the specified type. The `PWR_AttrName, PWR_AttrStat` pair determines the statistic that will be reported. For example, the user of this function might desire the maximum power used over a period of time one week prior to the current time. The user would specify the `id`, `id_type`, `PWR_ATTR_POWER` for the attribute and `PWR_STAT_MAX` for the statistic and populate the `start` and `stop` members of the `PWR_TimePeriod` structure appropriately. The times specified must be prior to the time when the function is called. The function returns the actual start and stop times if they differ from the times the user inputs. The implementation should return the time available time period that most closely matches the requested time period. The implementation determines the supported attribute combinations. The context of the calling user will determine if the user has the necessary privilege to access this information. This functionality assumes the system has a data retention capability exposed to the user.

```
int PWR_GetReportByID( PWR_Cntxt context,
                       const char* id,
                       PWR_ID id_type,
                       PWR_AttrName name,
                       PWR_AttrStat stat,
                       double* value,
                       PWR_TimePeriod* ReportTimes );
```

| **Argument(s)** | **Input and/or Output** | **Description** |
|---|---|---|
| PWR_Cntxt context | Input | The calling user's context which can be used to determine data access for individual role/user combinations. |
| const char* id | Input | The ID that the statistic will be collected for. |
| PWR_ID id_type | Input | The type of ID used to interpret the ID input. |
| PWR_AttrName name | Input | The name of the attribute the statistic will be based on. |
| PWR_AttrStat stat | Input | The desired statistic. |
| double* value | Output | Pointer to a double that will contain the statistic. |
| PWR_TimePeriod* ReportTimes | Input/Output | The user specified window for the report (start and stop times must be specified). |

| **Return Code(s)** | **Description** |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE (Function is implemented but call failed) |
| PWR_RET_NOT_IMPLEMENTED | Indicates that the combination of the attribute statistic pair and ID is not supported by this implementation. |

# Chapter 6

# Role/System Interfaces

This chapter includes the specifications for all of the Role/System pair interfaces depicted in figure 1.1 on page 15. Each interface section first outlines the purpose the interface serves. Core functionality for each interface is exposed through the attribute functions (see section 4.4). Each interface section includes a table of the supported attributes for that interface. The table contains the *suggested* attributes that the implementation should support for each interface. The implementation can choose to implement additional, some subset, or none of the attributes listed for that interface. As previously mentioned, the implementation must implement all attribute functions whether individual attributes are supported or not. If a particular attribute is not supported for that interface the implementation should return `PWR_RET_NOT_IMPLEMENTED`.

In addition to the attribute functions, other Core (Common) functions are included in this specification. Each individual interface section will enumerate the Core (Common) functions that the specification suggests are applicable for that interface (see chapter 4 for details regarding Core (Common) functions). Again, the implementation must implement these functions but may choose not to support them for a particular interface.

Each section also includes the High-Level (Common) functions that are applicable to that section (see chapter 6.10.3 for details regarding High-Level (Common) functions). These functions are functions that are applicable to more than one Role/System pair interface.

Finally, individual interface sections may also contain interface specific functions. These are functions that, at the time of their addition to the specification, are specific to one Role/System pair. This does not indicate that the function cannot be supported by an implementation for other Role/System pairs, only that the authors did not recognize a use for other interfaces at the time of addition to the specification.

## 6.1 Operating System, Hardware Interface

The Operating system/Hardware Interface is intended to be a low level interface that exposes power and energy relevant architecture features of the underlying hardware, such as the ability to measure and control power and energy characteristics of underlying components. In some cases this information will be abstracted for presentation to the application through the Application/Operating

System API interface (section 6.3) or the resource manager through the Resource Manager/Operating System API (section 6.5). While we have chosen the term `Operating system` as part of this interface name, we are not strictly implying that all interfaces described in this section should be limited to the domain of the operating system. Additionally, we are not implying that this interface requires specific privileges, although many low level operations require elevated privileges. Portions of the system software stack, like a runtime system, may use many of the interfaces described in this section.

### 6.1.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 4.4). The attribute functions in conjunction with the following attributes (Table 6.1) expose numerous measurement (get) and control (set) capabilities to the operating system.

Table 6.1: Operating System, Hardware - Supported Attributes

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_PSTATE | Set/Get | uint64_t | The current P-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE | Set/Get | uint64_t | The current C-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE_LIMIT | Set/Get | uint64_t | The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_SSTATE | Set/Get | uint64_t | The current S-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CURRENT | Get | double | Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call. |

Continued on next page

Table 6.1 – continued from previous page

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_VOLTAGE | Get | double | Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER | Get | double | Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER_LIMIT_MIN | Set/Get | double | Minimum power limit (floor, lower bound) for the specified object in watts. |
| PWR_ATTR_POWER_LIMIT_MAX | Set/Get | double | Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts. |
| PWR_ATTR_FREQ | Set/Get | double | The current operating frequency value for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MIN | Set/Get | double | Minimum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MAX | Set/Get | double | Maximum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_ENERGY | Get | double | The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value. |
| PWR_ATTR_TEMP | Get | double | The current temperature value for the specified object in degrees Celsius. |

### 6.1.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 4.2
  - ALL
- Group Functions - section 4.3
  - ALL
- Attribute Functions - section 4.4
  - ALL
- Metadata Functions - section 4.5
  - ALL
- Statistics Functions - section 4.6
  - ALL - for real time queries only

### 6.1.3 Supported High-Level (Common) Functions

### 6.1.4 Interface Specific Functions

#### Function Prototype PWR_StateTransitDelay()

The `PWR_StateTransitDelay` function returns the expected latency to transition between two valid states in nanoseconds. It is up to the vendor to provide accurate estimates for hardware. For example, P-state transitions could be given a single latency, even though some transitions might take less time (e.g., high voltage to lower voltage versus low to high). The desired state must be expressed using a `PWR_OperState` structure described in section 3.10 on page 31. This transition time may be a worst case latency time, and may be supplied by the hardware manufacturer (through the BIOS or other reporting mechanism). It is expected that this delay is an estimate of the time required to transition between states, not an estimate of the time that the core is unavailable for use (which may be a shorter interval than the time for the changes to take effect).

```
int PWR_StateTransitDelay( PWR_Obj obj,
                           PWR_OperState start_state,
                           PWR_OperState end_state,
                           PWR_Time *latency );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| `PWR_Ob obj` | Input | The object that the state transition would be applied to. |
| `PWR_OperState start_state` | Input | The state at the beginning of the transition. |
| `PWR_OperState end_state` | Input | The state at the end of the transition. |
| `PWR_Time *latency` | Output | Pointer to a double that will contain the transition latency in nanoseconds upon return. |

| Return Code(s) | Description |
|---|---|
| `PWR_RET_SUCCESS` | Upon SUCCESS |
| `PWR_RET_FAILURE` | Upon FAILURE |

## 6.2 Monitor and Control, Hardware Interface

The Monitor and Control/Hardware interface is targeted to support a critical function on HPC platforms (systems monitoring and management) often embodied in Reliability Availability and Serviceability (RAS) systems. RAS systems must evolve to measure and control power and energy relevant aspects of the system and serve this information and capability to administrators (Administrator/Monitor and Control Interface - section 6.7), resource managers (Resource Manager/Monitor and Control Interface - section 6.6), accounting (Accounting/Monitor and Control Interface - section 6.9) and users (User/Monitor and Control Interface - section 6.10). The Monitor and Control Interface serves more other roles than any other system in this specification. The base level functionality that is exposed through this interface is very similar to the Operating System/Hardware Interface (section 6.1) but the functional responsibilities of the role differ considerably. Some of the interfaces described in this specification imply data retention, or database, functionality. The monitor and control software (RAS system) is a prime candidate to serve this purpose. Low level power and energy data can be mined through the interfaces documented in this section and stored in raw or processed form in a database and made available for historic queries by other roles.

### 6.2.1 Supported Attributes

As in the Operating System/Hardware interface (section 6.1) a significant amount of functionality for this interface is exposed through the attribute functions (section 4.4). The attribute functions in conjunction with the following attributes (Table 6.2) expose numerous measurement (get) and control (set) capabilities to the monitor and control system.

Table 6.2: Monitor and Control, Hardware - Supported Attributes

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_PSTATE | Set/Get | uint64_t | The current P-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE | Set/Get | uint64_t | The current C-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE_LIMIT | Set/Get | uint64_t | The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable). |

Continued on next page

Table 6.2 – continued from previous page

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_SSTATE | Set/Get | uint64_t | The current S-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CURRENT | Get | double | Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_VOLTAGE | Get | double | Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER | Get | double | Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER_LIMIT_MIN | Set/Get | double | Minimum power limit (floor, lower bound) for the specified object in watts. |
| PWR_ATTR_POWER_LIMIT_MAX | Set/Get | double | Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts. |
| PWR_ATTR_FREQ | Set/Get | double | The current operating frequency value for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MIN | Set/Get | double | Minimum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MAX | Set/Get | double | Maximum operating frequency limit for the specified object in Hz (cycles per second). |

Continued on next page

Table 6.2 – continued from previous page

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_ENERGY | Get | double | The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value. |
| PWR_ATTR_TEMP | Get | double | The current temperature value for the specified object in degrees Celsius. |

### 6.2.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 4.2
  - ALL
- Group Functions - section 4.3
  - ALL
- Attribute Functions - section 4.4
  - ALL
- Metadata Functions - section 4.5
  - ALL
- Statistics Functions - section 4.6
  - ALL

### 6.2.3 Supported High-Level (Common) Functions

### 6.2.4 Interface Specific Functions

## 6.3 Application, Operating System Interface

The Application/Operating System Interface is intended to expose the appropriate level of information (measurement) and control to the application user or application library. This interface may also provide functionality appropriate for other levels of system software, such as a runtime system. The capabilities included in this interface concentrate on providing abstractions that allow an application or library to provide information that can be used to make intelligent decisions regarding performance, power and energy efficiency.

An important aspect of this interface is accommodating portable application (or library) code. Generalized concepts such as performance and sleep states that hardware can operate in are used rather than architecture specific concepts such as hardware P-States. The operating system, or privileged layer, is responsible for appropriately translating the abstracted information provided by the application layer into the hardware specific details necessary for accomplishing the desired functionality (or not). In essence the operating system, or privileged layer, acts as the hardware translator for the application.

### 6.3.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 4.4). The attributes functions in conjunction with the following attributes (Table 6.3) expose numerous measurement and control capabilities to the application, application libraries or possibly portions of runtime systems.

Table 6.3: Application, Operating System - Supported Attributes

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_POWER | Get | double | Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER_LIMIT_MIN | Set/Get | double | Minimum power limit (floor, lower bound) for the specified object in watts. |
| PWR_ATTR_POWER_LIMIT_MAX | Set/Get | double | Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts. |
| PWR_ATTR_FREQ | Set/Get | double | The current operating frequency value for the specified object in Hz (cycles per second). |

Continued on next page

Table 6.3 – continued from previous page

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_FREQ_LIMIT_MIN | Set/Get | double | Minimum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MAX | Set/Get | double | Maximum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_ENERGY | Get | double | The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value. |
| PWR_ATTR_TEMP | Get | double | The current temperature value for the specified object in degrees Celsius. |

### 6.3.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 4.2
  - ALL
- Group Functions - section 4.3
  - ALL
- Attribute Functions - section 4.4
  - ALL
- Metadata Functions - section 4.5
  - ALL
- Statistics Functions - section 4.6
  - ALL - for real time queries only

### 6.3.3 Supported High-Level (Common) Functions

### 6.3.4 Interface Specific Functions

#### Function Prototype PWR_AppTuningHint()

The PWR_AppTuningHint function is intended to be used by an application, or application library, to supply power relevant hints to the operating system (or a runtime layer). It is intentional that many of these hints do not directly imply that a power or energy adjustment will be made. The PWR_RegionHint hints are intended to be used by the application layer to indicate that it is entering a SERIAL, PARALLEL, COMPUTE (computation intensive) or COMMUNICATE, I/O or MEM_BOUND (communication intensive, I/O intensive or memory bound) region. The PWR_RegionHint type is described in section 3.11 on page 31. It is intended that these hints may be leveraged to provide some performance or power benefit, for example, a hint may indicate that an intensely parallel region is about to happen, this may motivate the proactive migration of tasks to an accelerator or preemptively speed up cooling fans to proactively deal with the thermal load.

PWR_RegionIntensity, described in section 3.11 on page 32 can be used for finer-grained hints than are possible with PWR_RegionHint. It is intended to allow for more explicit hints as to the intensity of the described region behavior. For example, it can be used to describe the intensity of a memory bound region, which can be utilized by the runtime or operating system in deciding what resources to allocate for a given power budget.

It is expected that the implementation will use these hints whenever possible to increase application performance while honoring energy/power targets or increase energy efficiency without incurring significant performance penalties. PWR_RegionIntensity may be set to PWR_REGION_INT_NONE if it is desirable for the operating system or a runtime to determine the intensity of resource usage dependent on the given hint. PWR_REGION_INT_NONE can also be used when the intensity of the described behavior is not known. This parameter may be ignored by the OS.

```
int PWR_AppTuningHint( PWR_Obj obj,
                       PWR_RegionHint hint,
                       PWR_RegionIntensity level );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj obj | Input | The object that the hint applies to. |
| PWR_RegionHint hint | Input | The hint corresponding to the code (behavioral) region being entered. |
| PWR_RegionIntensity level | Input | An abstraction of the intensity of the region. |

| Return Code(s) | Description |
|---|---|
| PWR_ERR_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NOT_IMPLEMENTED | Object does not support the requested operation |

### Function Prototype PWR_SetSleepStateLimit()

PWR_SetSleepStateLimit allows the application to request that, when possible, the OS restrict the deepest sleep state (e.g. C-state) that the hardware can enter. It is important to note that this function does not place the object in a sleep state, it only suggests to the Operating System (or privileged layer) that it limit the deepest possible sleep state that the object can enter. The operating system or hardware are responsible for determining when hardware should be put to sleep. This is not required to be honored by the OS or HW, but serves as a hint to the OS as to the latency that can be tolerated when transitioning between sleep and active states. As the application cannot typically control the entry of hardware into sleep states this function is meant to provide a method for an application to express its latency tolerance in an environment where resources may be put into sleep states without the application's knowledge.

Applications calling PWR_SetSleepStateLimit are expected to make use of the PWR_WakeUpLatency call on page 88 to provide information needed to determine the desired sleep state level. Sleep states must conform to the PWR_SleepState type in section 3.11 on page 32.

```
int PWR_SetSleepStateLimit( PWR_Obj obj,
                            PWR_SleepState state );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj obj | Input | The object to set the sleep state on. |
| PWR_SleepState state | Input | The sleep state to set as the maximum deepest sleep allowed. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NOT_IMPLEMENTED | Object does not support the requested operation |

### Function Prototype for PWR_WakeUpLatency()

The PWR_WakeUpLatency function returns a value in nanoseconds that corresponds to the time required to resume normal operation when transitioning out of a given sleep state. If the supplied

PWR_Obj does not support sleeping or the requested sleep state is not available then the function may return PWR_RET_FAILURE.

```
int PWR_WakeUpLatency( PWR_Obj obj,
                       PWR_SleepState state,
                       PWR_Time* latency );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj obj | Input | The object to query for latency. |
| PWR_SleepState state | Input | The sleep state to transition out of. |
| PWR_Time* latency | Output | The latency of the transition in nanoseconds. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NOT_IMPLEMENTED | Object does not support the requested operation |

*Advice to users: This function is useful when determining what sleep states can be exploited when knowledge of the length of time that certain operations (most likely remote ones) can be expected to take. Use of this function is intended to be paired with the SetSleepStateLimit function. Although users cannot use this function to place hardware into a sleep state, when used in conjunction with SetSleepStateLimit it can be used to suggest to an actor placing the hardware in a sleep state which state may be the most desirable. End of Advice to users.*

### Function Prototype PWR_RecommendSleepState()

This is a convenience function for cases in which an application's maximum tolerable latency is known for a given region and a deepest possible sleep state for use with the SetSleepStateLimit function is desired. Calling RecommendSleepState with the known latency will return the sleep state that has the closest latency to the desired value without exceeding it. Returned sleep states from this function conform to the PWR_SleepState type in section 3.11 on page 32.

```
int PWR_RecommendSleepState( PWR_Obj obj,
                             PWR_Time latency,
                             PWR_SleepState* state );
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj obj | Input | The object to set the sleep state on. |
| PWR_Time latency | Input | The amount of latency tolerable to the application in nanoseconds. |
| PWR_SleepState* state | Output | The deepest sleep state recommended to be used as a limit. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NOT_IMPLEMENTED | Object does not support the requested operation |

### Function Prototype for PWR_SetPerfState()

The PWR_SetPerfState function is used to request that an object change its performance level. The operating system is responsible for translating the abstracted PWR_PerfState value into an appropriate hardware-specific performance level (e.g. a CPU P-State). Setting the performance state of an object is not guaranteed to result in the requested change. The operating system may choose to ignore it or the hardware may not honor the request. The user should not expect that once a performance state has been set that it will not change in the future. Multiple actors may also set the performance state, including in some cases, remote actors.

```
int PWR_SetPerfState( PWR_Obj obj,
                      PWR_PerfState state);
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj obj | Input | The object to set the performance state on. |
| PWR_PerfState state | Input | The performance state to set the object to. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NOT_IMPLEMENTED | Object does not support the requested operation |

### Function Prototype for PWR_GetPerfState()

The PWR_GetPerfState function returns the performance state for any given object. The value that is returned is an abstracted value based on the real hardware state of the object that is mapped to the closest PWR_PerfState value. Objects must return PWR_RET_FAILURE if they do not support operating in different states.

```
int PWR_GetPerfState( PWR_Obj obj,
                      PWR_PerfState* state);
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj obj | Input | The object to get the current performance state of. |
| PWR_PerfState* state | Output | The performance state of the object. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NOT_IMPLEMENTED | Object does not support the requested operation |

### Function Prototype for PWR_GetSleepState()

The PWR_GetSleepState function returns the current sleep state for any given object.

```
int PWR_GetSleepState( PWR_Obj obj,
                       PWR_SleepState* state);
```

| Argument(s) | Input and/or Output | Description |
|---|---|---|
| PWR_Obj obj | Input | The object to get the current sleep state of. |
| PWR_PerfState* state | Output | The sleep state of the object. |

| Return Code(s) | Description |
|---|---|
| PWR_RET_SUCCESS | Upon SUCCESS |
| PWR_RET_FAILURE | Upon FAILURE |
| PWR_RET_NOT_IMPLEMENTED | Object does not support the requested operation |

## 6.4 User, Resource Manager Interface

The User/Resource Manger Interface is intended to support access to power and energy related information, specifically pertaining to jobs, relevant to an HPC user. This interface is similar to the User/Monitor and Control Interface (section 6.10) but in this case assumes that the Resource Manager has a data retention capability (database) available to query energy and statistics information based on job or user Id. The availability of this information is implementation dependent. Alternatively, if the Resource Manager does not have a database capability, the same interfaces are available to the user role through the User/Monitor and Control System Interface (section 6.10 which may provide this functionality.

### 6.4.1 Supported Attributes

The Power API specification does not currently recommend that any of the attributes be exposed to the user role. The implementation is free to expose any attribute they determine is useful to the user role without violating the specification.

### 6.4.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 4.2
  - ALL
- Group Functions - section 4.3
  - ALL
- Metadata Functions - section 4.5
  - ALL
- Statistics Functions - section 4.6
  - ALL - for historic queries only

### 6.4.3 Supported High-Level (Common) Functions

- Report Functions - section 5.1
  - ALL

### 6.4.4 Interface Specific Functions

## 6.5 Resource Manager, Operating System Interface

The Resource Manager/Operating System Interface is intended to access both low level and abstracted information from the operating system. Similar or additional information may be available from the monitor and control system (section 6.6) depending on the implementation. The resource manager is in a somewhat unique position of providing a range of functionality depending on the specific implementation. The resource manager role includes functionality such as batch schedulers and allocators as well as potential portions of tightly integrated runtime and launch systems. The resource manager may require fairly low level measurement information to make decisions and potentially store historic information for consumption by the user role (for example). The resource manager may also play a very large role in controlling power and energy pertinent functionally on both a application and platform basis in response to facility restrictions (power capping or energy aware scheduling for example).

### 6.5.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 4.4). The attribute functions in conjunction with the following attributes (Table 6.4) expose numerous measurement (get) and control (set) capabilities to the resource manager.

Table 6.4: Resource Manager, Operating System - Supported Attributes

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_PSTATE | Set/Get | uint64_t | The current P-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE | Set/Get | uint64_t | The current C-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE_LIMIT | Set/Get | uint64_t | The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_SSTATE | Set/Get | uint64_t | The current S-state for the object specified (typically processors but for use with other component types when applicable). |

Continued on next page

Table 6.4 – continued from previous page

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_POWER | Get | double | Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER_LIMIT_MIN | Set/Get | double | Minimum power limit (floor, lower bound) for the specified object in watts. |
| PWR_ATTR_POWER_LIMIT_MAX | Set/Get | double | Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts. |
| PWR_ATTR_FREQ | Set/Get | double | The current operating frequency value for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MIN | Set/Get | double | Minimum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MAX | Set/Get | double | Maximum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_ENERGY | Get | double | The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value. |
| PWR_ATTR_TEMP | Get | double | The current temperature value for the specified object in degrees Celsius. |

### 6.5.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 4.2
  - ALL
- Group Functions - section 4.3
  - ALL
- Attribute Functions - section 4.4
  - ALL
- Metadata Functions - section 4.5
  - ALL
- Statistics Functions - section 4.6
  - ALL

### 6.5.3 Supported High-Level (Common) Functions

### 6.5.4 Interface Specific Functions

## 6.6 Resource Manager, Monitor and Control Interface

The Resource Manager/Monitor and Control Interface is intended to access both low level and abstracted information from the monitor and control system (if available), much like the Resource Manager/Operating System Interface (section 6.5). The resource manager is in a somewhat unique position of providing a range of functionality depending on the specific implementation. The resource manager role includes functionality such as batch schedulers and allocators as well as potential portions of tightly integrated runtime and launch systems. The resource manager may require fairly low level measurement information to make decisions and potentially store historic information for consumption by the user role (for example). In contrast to the Resource Manager/-Operating System Interface (section 6.5) this interface includes the capability to mine information from the Monitor and Control system in situations where the Resource Manager does not retain historic data itself. The resource manager may also play a very large role in controlling power and energy pertinent functionally on both a application and platform basis in response to facility restrictions (power capping or energy aware scheduling for example).

### 6.6.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 4.4). The attribute functions in conjunction with the following attributes (Table 6.5) expose numerous measurement (get) and control (set) capabilities to the resource manager.

Table 6.5: Resource Manager, Monitor and Control - Supported Attributes

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_PSTATE | Set/Get | uint64_t | The current P-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE | Set/Get | uint64_t | The current C-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE_LIMIT | Set/Get | uint64_t | The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable). |

Continued on next page

Table 6.5 – continued from previous page

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_SSTATE | Set/Get | uint64_t | The current S-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_POWER | Get | double | Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER_LIMIT_MIN | Set/Get | double | Minimum power limit (floor, lower bound) for the specified object in watts. |
| PWR_ATTR_POWER_LIMIT_MAX | Set/Get | double | Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts. |
| PWR_ATTR_FREQ | Set/Get | double | The current operating frequency value for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MIN | Set/Get | double | Minimum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MAX | Set/Get | double | Maximum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_ENERGY | Get | double | The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value. |
| PWR_ATTR_TEMP | Get | double | The current temperature value for the specified object in degrees Celsius. |

### 6.6.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 4.2
  - ALL
- Group Functions - section 4.3
  - ALL
- Attribute Functions - section 4.4
  - ALL
- Metadata Functions - section 4.5
  - ALL
- Statistics Functions - section 4.6
  - ALL

### 6.6.3 Supported High-Level (Common) Functions

- Report Functions - section 5.1
  - ALL

### 6.6.4 Interface Specific Functions

## 6.7 Administrator, Monitor and Control Interface

The Administrator/Monitor and Control Interface is intended to expose administrator level measurement and control capabilities to the administrator role for the HPC platform. This interface assumes that the administrator role has elevated privileges. Additionally, the administrator is assumed to have access to all user role functionality documented in sections 6.10 and 6.4. A full complement of access to low level information is exposed through the attribute interface and other core level functions.

### 6.7.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 4.4). The attribute functions in conjunction with the following attributes (Table 6.6) expose numerous measurement (get) and control (set) capabilities to the administrator role.

Table 6.6: Monitor and Control, Hardware - Supported Attributes

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_PSTATE | Set/Get | uint64_t | The current P-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE | Set/Get | uint64_t | The current C-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE_LIMIT | Set/Get | uint64_t | The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_SSTATE | Set/Get | uint64_t | The current S-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CURRENT | Get | double | Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call. |

Continued on next page

Table 6.6 – continued from previous page

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_VOLTAGE | Get | double | Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER | Get | double | Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER_LIMIT_MIN | Set/Get | double | Minimum power limit (floor, lower bound) for the specified object in watts. |
| PWR_ATTR_POWER_LIMIT_MAX | Set/Get | double | Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts. |
| PWR_ATTR_FREQ | Set/Get | double | The current operating frequency value for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MIN | Set/Get | double | Minimum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MAX | Set/Get | double | Maximum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_ENERGY | Get | double | The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value. |
| PWR_ATTR_TEMP | Get | double | The current temperature value for the specified object in degrees Celsius. |

### 6.7.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 4.2
  - ALL
- Group Functions - section 4.3
  - ALL
- Attribute Functions - section 4.4
  - ALL
- Metadata Functions - section 4.5
  - ALL
- Statistics Functions - section 4.6
  - ALL

### 6.7.3 Supported High-Level (Common) Functions

- Report Functions - section 5.1
  - ALL

### 6.7.4 Interface Specific Functions

## 6.8 HPCS Manager, Resource Manager Interface

The HPCS Manager/Resource Manager Interface is intended to provide the necessary functionality for the HPCS Manager to implement policy via the Resource Manager. Policy information such as power caps (minimums or maximums), per user energy limits and traditional policies like node hours and priorities will all play a role in energy aware platform scheduling.

### 6.8.1 Supported Attributes

The Power API specification does not currently recommend that any of the attributes be exposed to the HPCS Manager role. The implementation is free to expose any attribute they determine is useful to the user role without violating the specification.

### 6.8.2 Supported Core (Common) Functions

### 6.8.3 Supported High-Level (Common) Functions

### 6.8.4 Interface Specific Functions

## 6.9 Accounting, Monitor and Control Interface

The Accounting/Monitor and Control Interface is intended to support access to power and energy related information regarding users, jobs and platform details to the accounting role. The accounting role differs from the user role in part by the elevated permissions this role will typically have. The accounting role includes interfaces to expose both a low-level interface via the attribute interface and higher level energy and statistics information through interface specific functions. The availability of historic information, critical to much of the accounting role, is dependent on the availability of the information in the Monitor and Control System which is implementation specific.

### 6.9.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 4.4). The attribute functions in conjunction with the following attributes (Table 6.7) expose numerous measurement (get) and control (set) capabilities to the accounting role.

Table 6.7: Accounting, Monitor and Control System - Supported Attributes

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_PSTATE | Set/Get | uint64_t | The current P-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE | Set/Get | uint64_t | The current C-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE_LIMIT | Set/Get | uint64_t | The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_SSTATE | Set/Get | uint64_t | The current S-state for the object specified (typically processors but for use with other component types when applicable). |

Continued on next page

Table 6.7 – continued from previous page

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_CURRENT | Get | double | Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_VOLTAGE | Get | double | Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER | Get | double | Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER_LIMIT_MIN | Set/Get | double | Minimum power limit (floor, lower bound) for the specified object in watts. |
| PWR_ATTR_POWER_LIMIT_MAX | Set/Get | double | Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts. |
| PWR_ATTR_FREQ | Set/Get | double | The current operating frequency value for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MIN | Set/Get | double | Minimum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MAX | Set/Get | double | Maximum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_ENERGY | Get | double | The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value. |

Continued on next page

Table 6.7 – continued from previous page

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_TEMP | Get | double | The current temperature value for the specified object in degrees Celsius. |

### 6.9.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 4.2
  - ALL
- Group Functions - section 4.3
  - ALL
- Attribute Functions - section 4.4
  - ALL
- Metadata Functions - section 4.5
  - ALL
- Statistics Functions - section 4.6
  - ALL

### 6.9.3 Supported High-Level (Common) Functions

- Report Functions - section 5.1
  - ALL

### 6.9.4 Interface Specific Functions

## 6.10 User, Monitor and Control Interface

The User/Monitor and Control Interface is intended to support access to power and energy information relevant to an HPC user. This interface is similar to the User/Resource Manager Interface (section 6.4) but exposes more low level information to the user through the Monitor and Control system, assuming the user has permission to access the information. The low level information exposed to the user role through this interface is primarily to support fine grained application analysis when available. The ability to mine energy and other statistics information based on job Id and user Id, included in this interface, assumes that a data retention capability is implemented in the Monitor and Control system. This is of course implementation dependent. Alternatively, if the Monitor and Control system does not have a database capability, the same interfaces are available to the user role through the User/Resource Manager Interface (section 6.4 which may provide this functionality.

### 6.10.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 4.4). The attribute functions in conjunction with the following attributes (Table 6.8) expose numerous measurement (get) and control (set) capabilities to the user role.

Table 6.8: User, Monitor and Control - Supported Attributes

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_PSTATE | Set/Get | uint64_t | The current P-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CSTATE | Set/Get | uint64_t | The current C-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_SSTATE | Set/Get | uint64_t | The current S-state for the object specified (typically processors but for use with other component types when applicable). |
| PWR_ATTR_CURRENT | Get | double | Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call. |

Continued on next page

Table 6.8 – continued from previous page

| Attribute | Set and/or Get | Type | Description |
|---|---|---|---|
| PWR_ATTR_VOLTAGE | Get | double | Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER | Get | double | Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call. |
| PWR_ATTR_POWER_LIMIT_MIN | Set/Get | double | Minimum power limit (floor, lower bound) for the specified object in watts. |
| PWR_ATTR_POWER_LIMIT_MAX | Set/Get | double | Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts. |
| PWR_ATTR_FREQ | Set/Get | double | The current operating frequency value for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MIN | Set/Get | double | Minimum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_FREQ_LIMIT_MAX | Set/Get | double | Maximum operating frequency limit for the specified object in Hz (cycles per second). |
| PWR_ATTR_ENERGY | Get | double | The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value. |
| PWR_ATTR_TEMP | Get | double | The current temperature value for the specified object in degrees Celsius. |

### 6.10.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 4.2
  - ALL
- Group Functions - section 4.3
  - ALL
- Attribute Functions - section 4.4
  - ALL
- Metadata Functions - section 4.5
  - ALL
- Statistics Functions - section 4.6
  - ALL

### 6.10.3 Supported High-Level (Common) Functions

- Report Functions - section 5.1
  - ALL

### 6.10.4 Interface Specific Functions

# Chapter 7

# Conclusion

The case for an HPC-community-adopted power API specification is compelling. The demand for computational cycles continues to increase, as does the expense to power the cycles. Hardware vendors are providing interfaces to power data and controls so that software can monitor usage and even control it. To maximize utilization of these ”knobs”, a portable interface layer allows multiple software products to code to a generic layer which can be translated by the individual hardware vendors. With this need in mind, the Power API defined herein sets out to address the following tenets.

**Very wide scope from facility to hardware component** This specification is not just limited to the hardware interfaces. The information from the hardware is the enabler for this API. However, the information is needed at many levels, from many different viewpoints. In [9] we identified a discrete set of unique actors (a.k.a. users, which can be software components) communicating via the API. In turn, these actors have interfaces with one or more systems within the scope of the API. The actor/system combinations represent the variety of viewpoints. For example, a batch job scheduler is more likely concerned about overall system and/or node power information, not the draw of a specific processor core or memory controller.

**Portability for software calling the API** By grouping the function calls by actor/system combination, we attempted to strike a balance between a totally non-intuitive, but generic get/put interface and one that is overly prescriptive by focusing on pre-identified and specific software packages. In addition to the actor/system calls, there is a set of calls to build the system “diagram” without having to rely on configuration files from a specific system type.

**Flexibility for implementer of an API** As this is a new area, the specification provides interfaces that are adaptable as hardware power technology evolves. The API is not based on any existing software-specific API. We can envision ways that interfaces such as RAPL, DVFS, NVML, BGQT/EMON, ACPI, the PAPI power interface, OpenMPI’s hwloc package, etc., etc. can become implementations for certain actor/system interfaces.

We strived to create a portable, implementable interface for power-aware computing. We welcome all suggestions and comments.

# References


[1] R. Bertran, Y. Sugawara, H. M. Jacobson, A. Buyuktosunoglu, and P. Bose. Application-level power and performance characterization and optimization on IBM Blue Gene/Q systems. In *IBM Journal of Research and Development*, volume 57, 2013.

[2] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[3] M. Brocanelli, Sen Li, Xiaorui Wang, and Wei Zhang. Joint management of data centers and electric vehicles for maximized regulation profits. In *Green Computing Conference (IGCC), 2013 International*, pages 1–10, June 2013.

[4] Hao Chen, Can Hankendi, Michael C. Caramanis, and Ayse K. Coskun. Dynamic Server Power Capping for Enabling Data Center Participation in Power Markets. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 122–129, Piscataway, NJ, USA, 2013. IEEE Press.

[5] Yuan Chen, D. Gmach, C. Hyser, Zhikui Wang, C. Bash, C. Hoover, and S. Singhal. Integrated management of application performance, power and cooling in data centers. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 615–622, April 2010.

[6] Yiannis Georgiou. Energy Accounting and Control on HPC clusters, November 2013. `http://perso.ens-lyon.fr/laurent.lefevre/greendayslille/greendayslille_Yiannis_Georgiou.pdf`.

[7] Yiannis Georgiou, Thomas Cadeau, David Glesser, Danny Auble, Morris Jette, and Matthieu Hautreux. Energy Accounting and Control with SLURM Resource and Job Management System. In Mainak Chatterjee, Jian-Nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *ICDCN*, volume 8314 of *Lecture Notes in Computer Science*, pages 96–118. Springer, 2014.

[8] Ivar Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[9] James H Laros, Suzanne M Kelly, Steven Hammond, Ryan Elmore, and Kristin Munch. Power/Energy Use Cases for High Performance Computing. Internal SAND Report SAND2013-10789. `https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/UseCase-powapi.pdf`.

[10] Zhenhua Liu, Yuan Chen, Cullen Bash, Adam Wierman, Daniel Gmach, Zhikui Wang, Manish Marwah, and Chris Hyser. Renewable and Cooling Aware Workload Management for Sustainable Data Centers. *SIGMETRICS Perform. Eval. Rev.*, 40(1):175–186, June 2012.

[11] Bryan Mills, Ryan E. Grant, Kurt B. Ferreira, and Rolf Riesen. Evaluating Energy Savings for Checkpoint/Restart. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, E2SC '13, pages 6:1–6:8, New York, NY, USA, 2013. ACM.

[12] D.K. Newsom, S.F. Azari, A. Anbar, and T. El-Ghazawi. Locality-aware power optimization and measurement methodology for PGAS workloads on SMP clusters. In *Green Computing Conference (IGCC), 2013 International*, pages 1–10, June 2013.

[13] J.U. Patel, S.J. Guercio, A.E. Bruno, M.D. Jones, and T.R. Furlani. Implementing green technologies and practices in a high performance computing center. In *Green Computing Conference (IGCC), 2013 International*, pages 1–8, June 2013.

[14] P.V. Ramakrishna, G. Kaushik, K.L. Sudhakar, G. Thiagarajan, and A. Sivasubramaniam. Online system for energy assessment in large facilities - Methodology amp; A real-world case study. In *Green Computing Conference (IGCC), 2013 International*, pages 1–9, June 2013.

[15] Geri Schneider and Jason Winters. *Apply Use Cases: A Practical Guide, Second Edition*. Addison-Wesley, 2001.

[16] Hayk Shoukourian, Torsten Wilde, Axel Auweter, and Arndt Bode. Monitoring power data: A first step towards a unified energy efficiency evaluation toolset for hpc data centers. *Environmental Modeling  Software*, 2013.

[17] Tiffany Trader. Green Power Management Deep Dive. *Green Computing Report*, June 2013. `http://www.greencomputingreport.com/gcr/2013-06-05/green_power_management_deep_dive.html`.

[18] Abhinav Vishnu, Shuaiwen Song, Andres Marquez, Kevin Barker, Darren Kerbyson, Kirk Cameron, and Pavan Balaji. Designing energy efficient communication runtime systems: a view from PGAS models. *The Journal of Supercomputing*, 63(3):691–709, 2013.

[19] Sean Wallace, Venkatram Vishwanath, Susan Coghlan, John Tramm, Zhiling Lan, and Michael E. Papka. Application Power Profiling on IBM Blue Gene/Q. In *Proceedings of 2013 International Conference on Cluster Computing*. IEEE, 2013.

[20] V.M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. Measuring Energy and Power with PAPI. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 262–268, Sept 2012.

[21] Xu Yang, Zhou Zhou, Sean Wallace, Zhiling Lan, Wei Tang, Susan Coghlan, and Michael E. Papka. Integrating Dynamic Pricing of Electricity into Energy Aware Scheduling for HPC Systems. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 60:1–60:11, New York, NY, USA, 2013. ACM.

# Appendix A

# Topics Under Consideration for Future Versions

The following topics are either currently in active discussion or are planned to be addressed in future versions of the specification. In some cases it will be necessary to solicit additional feedback from the community to ensure we properly address the issue in future versions.

- **Coexistence of Implementations** - One of the driving questions for this future work is - how does one implementation interface with another? It is possible, even likely that an implementor will focus on implementing a portion or portions of the specification. This begs the question of how does implementation A interact with implementation B? Further, what role does the specification play in driving this interaction? We intend to work closely with the community to sort out this issue and document the appropriate guidance in the next version of the specification.
- **Language Bindings** - Some roles, system administrator for example, more commonly interface with the platform through shells, shell scripting or other interpretive languages like Perl or Python. We will investigate adding some or all of these capabilities, via specification and possibly prototypes, in future versions of the standard.
  - The next version of the specification will include a complete Python specification of all existing functions modified appropriately for the Python language
- **User Guide** - The addition of a user guide could provide additional useful information to both users and implementors. The addition of a users guide will be considered and if realized will accompany subsequent releases of the specification.
- **Hypothetical System Example** - We are considering creating a hypothetical system example to use to discuss and clarify concepts and higher level use cases. This will likely be included in the User Guide.
- **Required versus Optional, or Quality of Implementation** - We plan to clarify and document more precisely what portions of the specification are required to be implemented, what portions are optional and the definition of a quality implementation. This topic is complicated by the fact that implementors are free to implement portions of the specification.
  - Some progress has been made on this topic for version 1.1 but additional work is required.
- **Policies** - Security policies, priority of operations and privileges need to be further vetted and specified when appropriate. This topic has a large amount of intersection with the *Coex-*

*istence of Implementations* topic and will be considered jointly.

- **Unit Tests** - Development of a unit test infrastructure is under consideration, possibly to be associated with our prototype which will be released open source at a later date. Unit tests might also be a way for the implementation community to assure interaction between implementations of portions of the specification that will be required to work together.
- **User Supplied Functions** - We intend to investigate adding the ability for a user to supply a function for the purposes of generating a statistic, for example.
- **Multiple Platform Support** - Currently the specification only considers operation on a single platform. There is nothing preventing supporting multiple platforms and exposing multiple platforms in a single context in future versions. This will be considered for the next release in conjunction with the Coexistence of Implementation issue.
- **Generation Counter** - We intend to consider the addition of a generation counter capability to be used in conjunction with counters that have the potential for roll over. The generation counter could be used to inform the user that this has taken place. This concept likely has additional utility which is what will be explored for future releases of the specification. Target: 1.X - Implementation should handle overflow internally
- **Time Conversion/Overflow** - Time conversion convenience functions are being considered to convert between `PWR_Time` values and POSIX-compatible time representations. Included in this will be methods of detecting overflow during time value arithmetic.
- **Context Refresh** - We are considering adding the ability to refresh a context int he case of a long lived context such as one that is used by a persistent daemon. Yet to be resolved is what happens to existing pointers, more specifically what happens when the user has a pointer to an object that no longer exists after the refresh, or if this can happen.
- **Move Transition Latencies to Metadata** - Currently P-state and C-state transition latencies are returned by `PWR_StateTransitDelay`. It may be cleaner to move this functionality to a generalized metadata interface. We will consider making this change in a future version of the specification.
- **Enhanced Support for ACPI 5.0** - Collaborative Processor Performance Control and Continuous Performance Control are currently not supported. Support will require new attributes and some function calls to allow for the flexible mechanisms provided in the ACPI 5.0 specification to allow expression of desired performance on a sliding, abstract unit-less scale. ACPI 5.0 also supports gathering statistics about the delivery of given performance values and the time spent in certain states, which we intend to address. We anticipate adding this support alongside the P-state and C-state functionality already in the Power API in a future version of the specification.
- **User/Resource Manager Interfaces** - Work needs to be done in this area but is best accomplished in collaboration with resource manager, work load manager experts. We hope to include standard interfaces for the user to query this system in future versions of the specification.
  - Work has begun to develop general report and information mining capabilities
- **HPCS Manager to Resource Manager Interface** - This interface clearly needs some work. Again it seems that this would benefit greatly from collaborative efforts.
  - Work has begun to develop general report and information mining capabilities

# Appendix B

# Change Log

The following list contains changes to version 1.1a of the specification.

- PWR_ID - Changed - PWR_USER_ID, PWR_JOB_ID and PWR_RUN_ID to PWR_ID_-USER, PWR_ID_JOB and PWR_ID_RUN respectively. This change was done in order to bring the PWR_AttrStat enum into line with existing naming conventions in the specification.
- PWR_ID - Changed - PWR_ID_SPECIFIED = -2 was changed to PWR_ID_NOT_SPECIFIED = -2 in order to better represent the purpose of the value.
- PWR_GrpUnion() - Changed - arguments were misnamed as PWR_Grp group, PWR_Grp group, they are now changed to PWR_Grp group1 and PWR_Grp group2.
- PWR_GrpIntersect() - Changed - arguments were misnamed as PWR_Grp group, PWR_Grp group, they are now changed to PWR_Grp group1 and PWR_Grp group2.
- PWR_GrpDifference() - Changed - arguments were misnamed as PWR_Grp group, PWR_-Grp group, they are now changed to PWR_Grp group1 and PWR_Grp group2.
- PWR_CntxtGetGrpByName() - Clarified - The source of the group names to be used with this function was clarified, they are to be provided in vendor documentation. Further notes have been added to make the user aware of the fact that due to this being dependent on individual vendors, it is by nature non-portable.
- PWR_StatGetReduce() - Changed - The return type of this function was a PWR_Stat when it should have been an int, it now returns an int.
- PWR_StatDestroy() - Changed - The return type of this function was a PWR_Stat when it should have been an int, it now returns an int.
- PWR_GetReportByID() - Changed - The function prototype was missing a comma after the *value argument, this typo was fixed.
- PWR_WakeUpLatency() - Changed - The function prototype was missing a comma after the PWR_SleepState state argument, this typo was fixed.

- PWR_RecommendedSleepState() - Changed - The function prototype was missing a comma after the PWR_Time latency argument, this typo was fixed.
- PWR_RecommendedSleepState() - Changed - PWR_Obj variable name was missing, this was added as ”obj”.

The following list contains changes to version 1.1 of this specification.

- PWR_GrpCreate() - Changed - This function no longer requires a user specified name parameter. The utility of the user specifying a name for user created groups was determined to be of little value.
- PWR_GrpDuplicate() - Changed - This function no longer requires a user specified name. The utility of the user specifying a name for user created groups was determined to be of little value.
- PWR_GrpGetName() - Removed - This function is no longer necessary due to the changes made to PWR_GrpCreate() and PWR_GrpDuplicate().
- PWR_GrpUnion() - Added - New function to create a new group which is the union of two existing groups
- PWR_GrpIntersection() - Added - New function to create a new group which is the Intersection of two existing groups
- PWR_GrpDifference() - Added - New function to create a new group which is the difference of two existing groups
- High Level (Common) Functions - Added - This chapter will contain any high level functions that are common to multiple Role/System interfaces.
- Report Functions - Added - This section of chapter High Level Functions will contain any high level report generating functions that are common to multiple Role/System interfaces.
- PWR_GetStatByID() - Changed - This function has been changed to PWR_GetReportByID() and added to the Report Functions section of the High Level (Common) Functions chapter. It requires an additional parameter of the users context. The statTimes parameter has been changed to ReportTimes and is now of type TimePeriod. The id_type parameter has been changed to type PWR_ID.
- PWR_GetStatByUser() - Removed - This function was considered an unnecessary abstraction of PWR_GetStatByID() which has been changed to PWR_GetReportByID().
- PWR_GetStatByJob() - Removed - This function was considered an unnecessary abstraction of PWR_GetStatByID() which has been changed to PWR_GetReportByID().
- PWR_GetEnergyByID() - Removed - This function was considered an unnecessary abstraction of PWR_GetStatByID() which has been changed to PWR_GetReportByID().

- PWR_GetEnergyByUser() - Removed - This function was considered an unnecessary abstraction of PWR_GetStatByID() which has been changed to PWR_GetReportByID().
- PWR_GetEnergyByJob() - Removed - This function was considered an unnecessary abstraction of PWR_GetStatByID() which has been changed to PWR_GetReportByID().
- PWR StatGetReduce() - Added - New functionality to preform a reduction (Min, Max, Avg, etc.) of a requested statistic.
- PWR_RET_FAILURE - Clarified - Noted that a failure return can be a partial failure, not necessarily a total failure. This is useful for PWR_GrpSetValues and PWR_GrpGetValues.
- PWR_StateTransitDelay() - Clarified - Removed a discussion on what expected values were to be as it was not true for all platforms.
- PWR_OperState - Changed - Updated PWR_OperState to use uint64_t types instead of ints.
- PWR_MD_TS_LATENCY - Typo - Accidentally listed as PWR_MD_LATENCY in the big list of attributes
- PWR_CntxtGetEntryPoint() - Clarified - Corrected description of function, incorrect in version 1.0
- PWR_StatID - Changed - Typedef changed to more generic PWR_ID
- PWR_StatTime - Changed - Typedef changed to more generic PWR_TimePeriod
- PWR_StatGetValue() - Changed - statTimes parameter now is of time PWR_TimePeriod
- PWR_StatGetValues() - Changed - statTimes parameter now is of time PWR_TimePeriod
- PWR_StatStart() - Changed - The stat parameter is now statObj, more descriptive
- PWR_StatClear() - Changed - The stat parameter is now statObj, more descriptive
- PWR_StatGetValues() - Changed - The stat parameter is now statObj, more descriptive
- Interfaces - Changed - Now called Role/System Interfaces, clarified description
- PWR_ATTR_THROTTLED_TIME - Added - New attribute to get the cumulative time throttled in nanoseconds.
- PWR_ATTR_THROTTLED_COUNT - Added - New attribute to get the cumulative count of throttle events.
- Attribute Get Functions - Changed - Allow NULL to be passed in for timestamp arguments if no timestamp is required.
- Attribute Get and Set Functions - Clarified - Clarified where values are stored at in input and output `void *` arrays. Previously the offset that value `i` is stored at was unspecified.

- Argument Types - Changed - Changed all occurrences of `int` to `uint64_t`.
- Argument Types - Changed - Changed all occurrences of `float` to `double`.
- PWR_Status - Clarified - Clarified that `PWR_Status` objects only contain the status of failed operations. The status of successful operations is not included.
- All functions that return a PWR_Status - Clarified that all operations requested by the caller are attempted at least once (i.e., the PowerAPI does not stop at the first error). Failed operations have their status returned in the `PWR_Status` object. All other operations succeeded.
- PWR_MAJOR_VERSION - Added - Added a compile time constant indicating the major version of the Power API version supported by the implementation.
- PWR_MINOR_VERSION - Added - Added a compile time constant indicating the minor version of the Power API version supported by the implementation.
- PWR_RET_WARN_NOT_OPTIMIZED - Added - New warning return code to indicate that the operation requested was not optimized.
- All Enumeration Types - Added - For each enumeration, added values for `INVALID`, `NOT_SPECIFIED`, and the count of values defined by the enum.
- PWR_ATTR_POWER_MIN - Changed - Renamed PWR_ATTR_POWER_MIN attribute to PWR_ATTR_POWER_LIMIT_MIN and clarified description.
- PWR_ATTR_POWER_MAX - Changed - Renamed PWR_ATTR_POWER_MAX attribute to PWR_ATTR_POWER_LIMIT_MAX and clarified description.

# DISTRIBUTION:

| | | |
|---|---|---|
| 1 | MS 1319 | James A. Ang, 1422 |
| 1 | MS 1319 | All Staff, 1422 |
| 1 | MS 1319 | Ron B. Brightwell, 1423 |
| 1 | MS 1319 | All Staff, 1423 |
| 1 | MS 0899 | Reports Management sanddocs@sandia.gov, 5936 |
| 1 | MS 0899 | Technical Library, 9536 (electronic copy) |

Sandia National Laboratories